The art of the language VM, or Machine-generating virtual machine code, or Almost zero overhead with almost zero assembly, or My virtual machine is faster than yours

Luca Saiu
positron@gnu.org
http://ageinghacker.net

GNU Project

GNU Hackers' Meeting 2017
Knüllwald-Niederbeisheim, Germany
August 25th 2017

Introduction and history

My main long-term project is GNU epsilon. It's a programming language, meant to be efficient, but:

- very "dynamic" in certain execution phases
- written in itself, bootstrapped

— Too slow.

So I wrote a canonical threaded-code VM.

- speedup 4-6x

— Too little.

So I made a separate repository to experiment with language VMs.

- tried techniques from scientific papers (many by Anton Ertl and the other GForth people)
- added ideas of my own

— It got completely out of hand.

A new project, independent from epsilon.

Introduction and history

My main long-term project is GNU epsilon. It's a programming language, meant to be efficient, but:

- very "dynamic" in certain execution phases
- written in itself, bootstrapped

— Too slow.

So I wrote a canonical threaded-code VM.

- speedup 4-6x

— Too little.

So I made a separate repository to experiment with language VMs.

- tried techniques from scientific papers (many by Anton Ertl and the other GForth people)
- added ideas of my own

— Still not completely out of hand.

A new project, independent from epsilon.

Introduction and history

My main long-term project is GNU epsilon. It's a programming language, meant to be efficient, but:

- very "dynamic" in certain execution phases
- written in itself, bootstrapped

— Too slow.

So I wrote a canonical threaded-code VM.

- speedup 4-6x

— Too little.

So I made a separate repository to experiment with language VMs.

- tried techniques from scientific papers (many by Anton Ertl and the other GForth people)
- added ideas of my own
- it got completely out of hand

A new project, independent from epsilon.

Introduction and history

My main long-term project is GNU epsilon. It's a programming language, meant to be efficient, but:

- very "dynamic" in certain execution phases
- written in itself, bootstrapped

— Too slow.

So I wrote a canonical threaded-code VM.

- speedup 4-6x

— Too little.

So I made a separate repository to experiment with language VMs.

- tried techniques from scientific papers (many by Anton Ertl and the other GForth people)
- added ideas of my own
- it got completely out of hand

A new project, independent from epsilon.

Why you should care

Interpreters are common:

- programming languages
- application scripting
- shells
- regular expressions...

- We are getting used to unacceptably bad performance.

I will present my new software, but first I need to describe the problem it solves. This will take a while.

Why you should care

Interpreters are common:

- programming languages
- application scripting
- shells
- regular expressions...

- We are getting used to unacceptably bad performance.

I will present my new software, but first I need to describe the problem it solves. This will take a while.

Why you should care

Interpreters are common:

- programming languages
- application scripting
- shells
- regular expressions. . .

- We are getting used to unacceptably bad performance.

I will present my new software, but first I need to describe the problem it solves. This will take a while.

Our running example — at first in C

Count down from two billion (here meaning $2 \cdot 10^9$):

C

```
int
main (void)
{
  long i;
  for (i = 2000000000; i != 0; i --)
    /* Do nothing */;
  return 0;
}
```

. . . does this program really count down?

Our running example — at first in C

Count down from two billion (here meaning $2 \cdot 10^9$):

C

```
int
main (void)
{
  long i;
  for (i = 2000000000; i != 0; i --)
    /* Do nothing */;
  return 0;
}
```

. . . does this program really count down?

Our running example — at first in C, now actually counting

Count down from $2 \cdot 10^9$ *without optimizing away the entire loop*:

C (with GNU extensions)

```
int
main (void)
{
  long i;
  for (i = 2000000000; i != 0; i --)
    asm volatile ("" : : "r" (i)); // pretend to use i
  return 0;
}
```

(We still want most GCC optimizations!)

[Demo: the down-counter in a few languages]

Our running example — at first in C, now actually counting

Count down from $2 \cdot 10^9$ *without optimizing away the entire loop*:

C (with GNU extensions)

```
int
main (void)
{
  long i;
  for (i = 2000000000; i != 0; i --)
    asm volatile ("" : : "r" (i)); // pretend to use i
  return 0;
}
```

(We still want most GCC optimizations!)

[Demo: the down-counter in a few languages]

Our running example — at first in C, now actually counting

Count down from $2 \cdot 10^9$ *without optimizing away the entire loop*:

C (with GNU extensions)

```
int
main (void)
{
  long i;
  for (i = 2000000000; i != 0; i --)
    asm volatile ("" : : "r" (i)); // pretend to use i
  return 0;
}
```

(We still want most GCC optimizations!)

[Demo: the down-counter in a few languages]

You can play with the sources

I will (quickly) show some interpreters written in C.

In case you want to play with the examples yourself, the little programs I'm showing here are on my server:

`http://ageinghacker.net/ghm-2017`

These are naïf C programs showing how interpreters work; the C files in `c-examples/` are *not* part of my new project.

How simple interpreters work

The interpreted program is a data structure in memory.
"find the next point in the interpreted program, execute it, repeat from start"

How to *dispatch* ["dispatch": moving from a VM program point to another]:

- Abstract Syntax Tree (AST) interpreters
- Linear programs
  - `switch` dispatching
  - direct threading
  - ...

How to access data:

- associative data structures (alists, hash tables)
- VM registers
- stacks

How simple interpreters work

The interpreted program is a data structure in memory.
"find the next point in the interpreted program, execute it, repeat from start"

How to *dispatch* ["dispatch": moving from a VM program point to another]:

- Abstract Syntax Tree (AST) interpreters
- Linear programs
  - `switch` dispatching
  - direct threading
  - ...

How to access data:

- associative data structures (alists, hash tables)
- VM registers
- stacks

How simple interpreters work

The interpreted program is a data structure in memory.
"find the next point in the interpreted program, execute it, repeat from start"

How to *dispatch* ["dispatch": moving from a VM program point to another]:

- Abstract Syntax Tree (AST) interpreters
- Linear programs
  - `switch` dispatching
  - direct threading
  - ...

How to access data:

- associative data structures (alists, hash tables)
- VM registers
- stacks

Our down-counter as an Abstract Syntax Tree

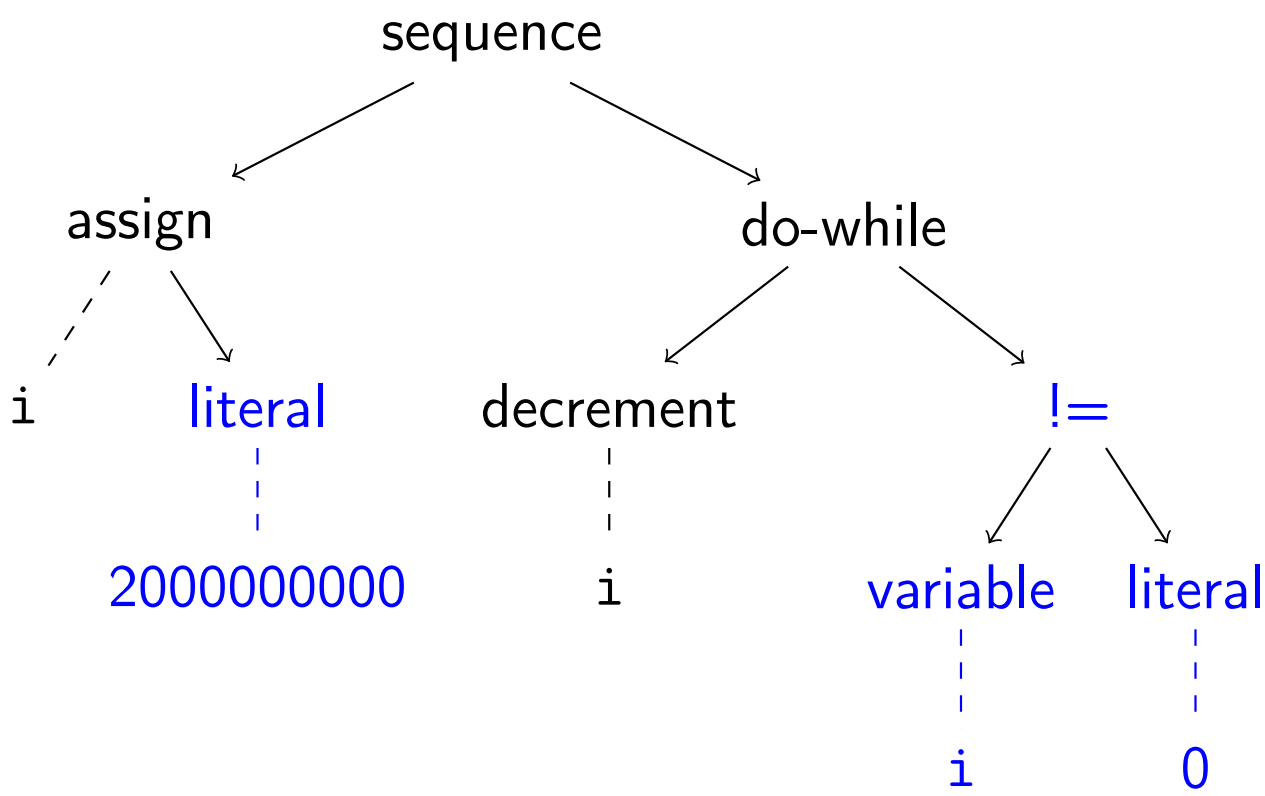
```
i := 2000000000;
do
  decrement i;
while i != 0;
```


A program is an Abstract Syntax Tree data structure in memory: heap-allocated `structs` and `unions` with lots of pointers. Each node has an `enum` field to distinguish its kind.

[Blue: expression node; dashed line: child is a `struct` field of parent; black arrow: parent contains pointer to child.]

Our down-counter as an Abstract Syntax Tree

```
i := 2000000000;
do
  decrement i;
while i != 0;
```

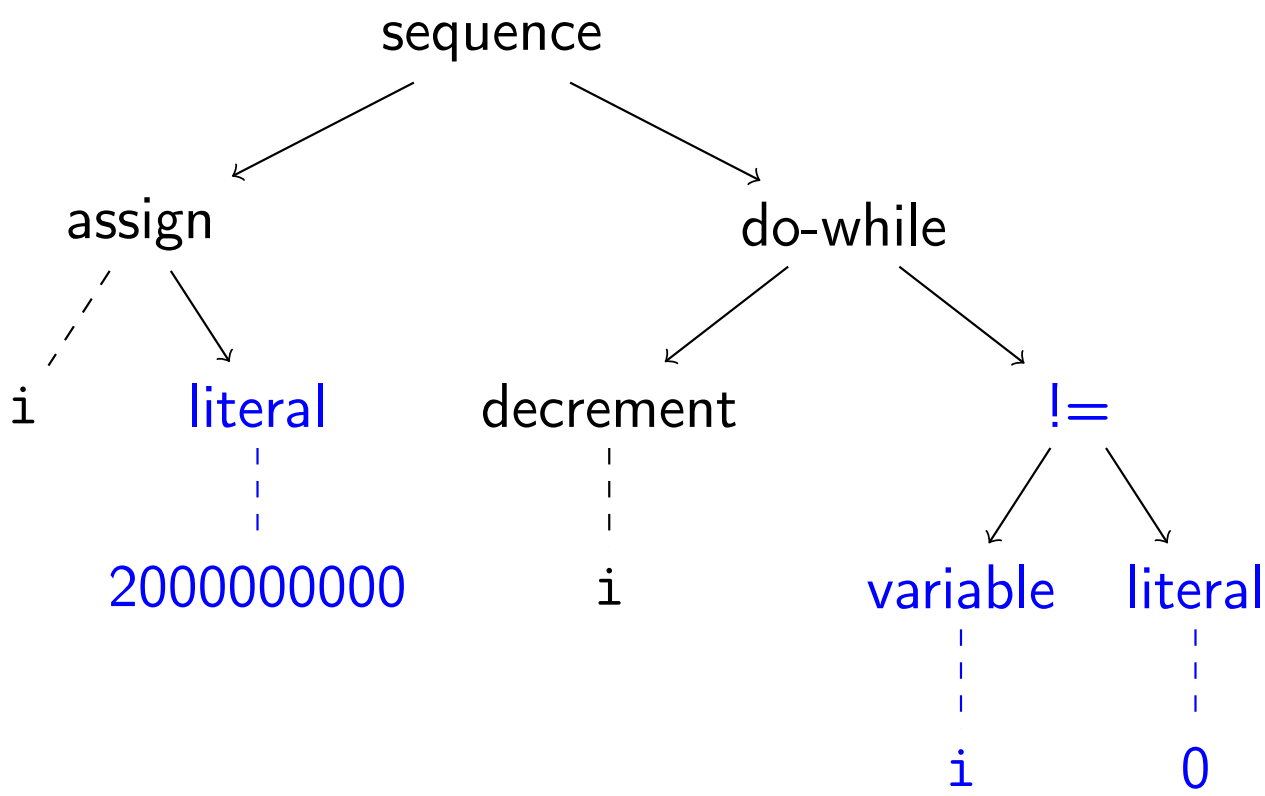


A program is an Abstract Syntax Tree data structure in memory: heap-allocated `structs` and `unions` with lots of pointers. Each node has an `enum` field to distinguish its kind.

[Blue: expression node; dashed line: child is a `struct` field of parent; black arrow: parent contains pointer to child.]

Our down-counter as an Abstract Syntax Tree

```
i := 2000000000;
do
  decrement i;
while i != 0;
```



A program is an Abstract Syntax Tree data structure in memory: heap-allocated `structs` and `unions` with lots of pointers. Each node has an `enum` field to distinguish its kind.

[Blue: expression node; dashed line: child is a `struct` field of parent; black arrow: parent contains pointer to child.]

Abstract Syntax Tree interpreter: expression

As each complex AST has sub-ASTs recursion is natural. AST data structures are easy to define in Lisp and ML, a little less pretty in C.

```
long
interpret_expr (const struct expr *e, const long *vars) {
  switch (e->expr_case) {
    case expr_variable:
      return vars [e->var_index];
    case expr_constant:
      return e->cnst;
    case expr_is_different:
      return (   interpret_expr (e->sub1, vars)
              != interpret_expr (e->sub2, vars));
    default:
      error ();
  }
}
```

Abstract Syntax Tree interpreter: statement

```
void interpret_stmt (const struct stmt *s, long *vars) {
  switch (s->stmt_case) {
  case stmt_sequence:
    interpret_stmt (s->sub1, vars);
    interpret_stmt (s->sub2, vars);
    break;
  case stmt_assign:
    vars [s->var_index] = interpret_expr (s->assigned_expr, vars);
    break;
  case stmt_decrement:
    vars [s->var_index] --;
    break;
  case stmt_dowhile:
    interpret_stmt (s->body, vars);
    if (interpret_expr (s->guard, vars))
      interpret_stmt (s, vars);
    break;
  default: error ();
  }
}
```

AST interpreter performance

- pointer chasing (load latency $\sim 3\tau$ on L1d hit!)

AST interpreter performance

- pointer chasing (load latency $\sim 3\tau$ on L1d hit!)
- many conditionals, often multi-way (mispredict penalty $\sim 15\tau$, per conditional branch!)

AST interpreter performance

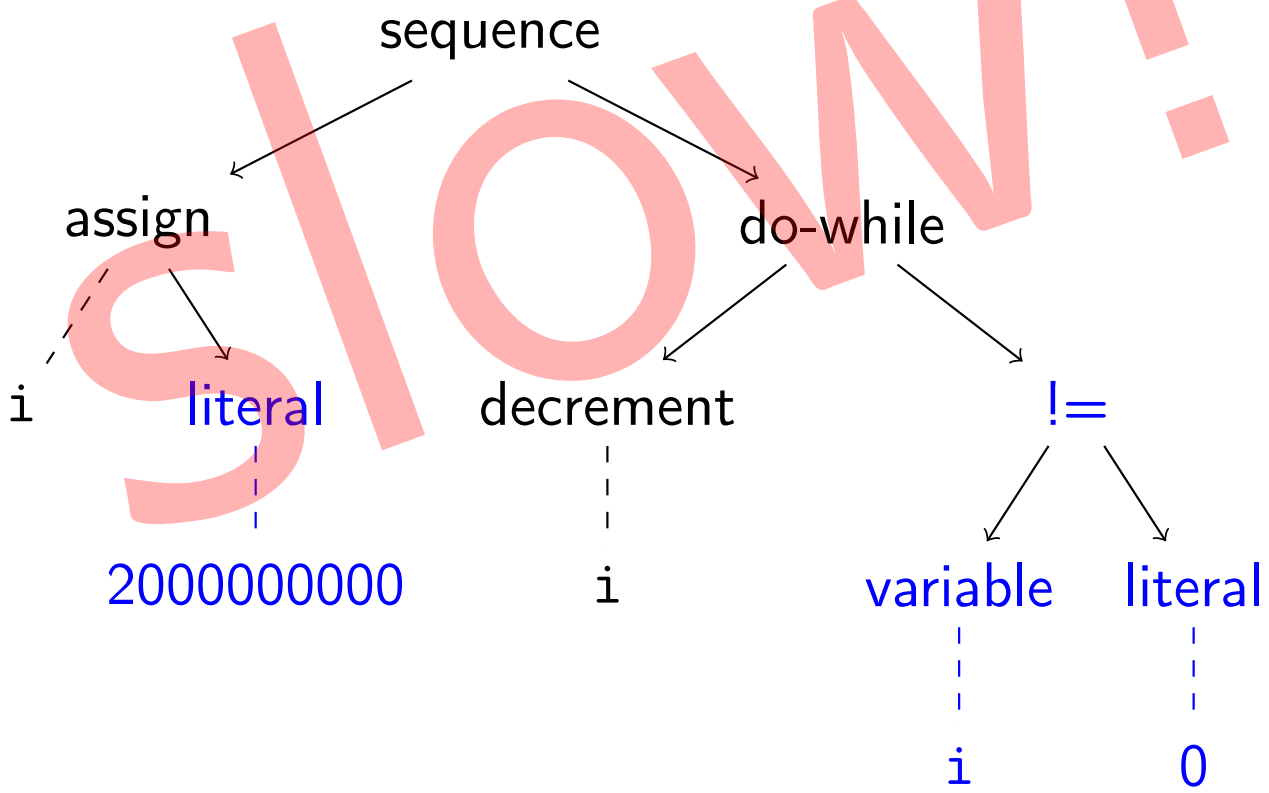
- pointer chasing (load latency $\sim 3\tau$ on L1d hit!)
- many conditionals, often multi-way (mispredict penalty $\sim 15\tau$, per conditional branch!)
- variable lookup slow (not shown in my sample code before)

AST interpreter performance

- pointer chasing (load latency $\sim 3\tau$ on L1d hit!)
- many conditionals, often multi-way (mispredict penalty $\sim 15\tau$, per conditional branch!)
- variable lookup slow (not shown in my sample code before)
- recursion, often non-tail

AST interpreter performance

- pointer chasing (load latency $\sim 3\tau$ on L1d hit!)
- many conditionals, often multi-way (mispredict penalty $\sim 15\tau$, per conditional branch!)
- variable lookup slow (not shown in my sample code before)
- recursion, often non-tail



A good language to interpret

What is normally called a language "Virtual Machine" is an interpreter for a lower-level linear program:

- the program to interpret is stored as a contiguous array in hardware memory
- no nesting: no statements with sub-statements or expressions with sub-expressions
- no expressions, no variables
- assembly-like feel: registers or stacks, explicit jumps

I'll show you a linear-program interpreter written in C.

A good language to interpret

What is normally called a language "Virtual Machine" is an interpreter for a lower-level linear program:

- the program to interpret is stored as a contiguous array in hardware memory
- no nesting: no statements with sub-statements or expressions with sub-expressions
- no expressions, no variables
- assembly-like feel: registers or stacks, explicit jumps

I'll show you a linear-program interpreter written in C.

The down-counter as a linear program to be interpreted

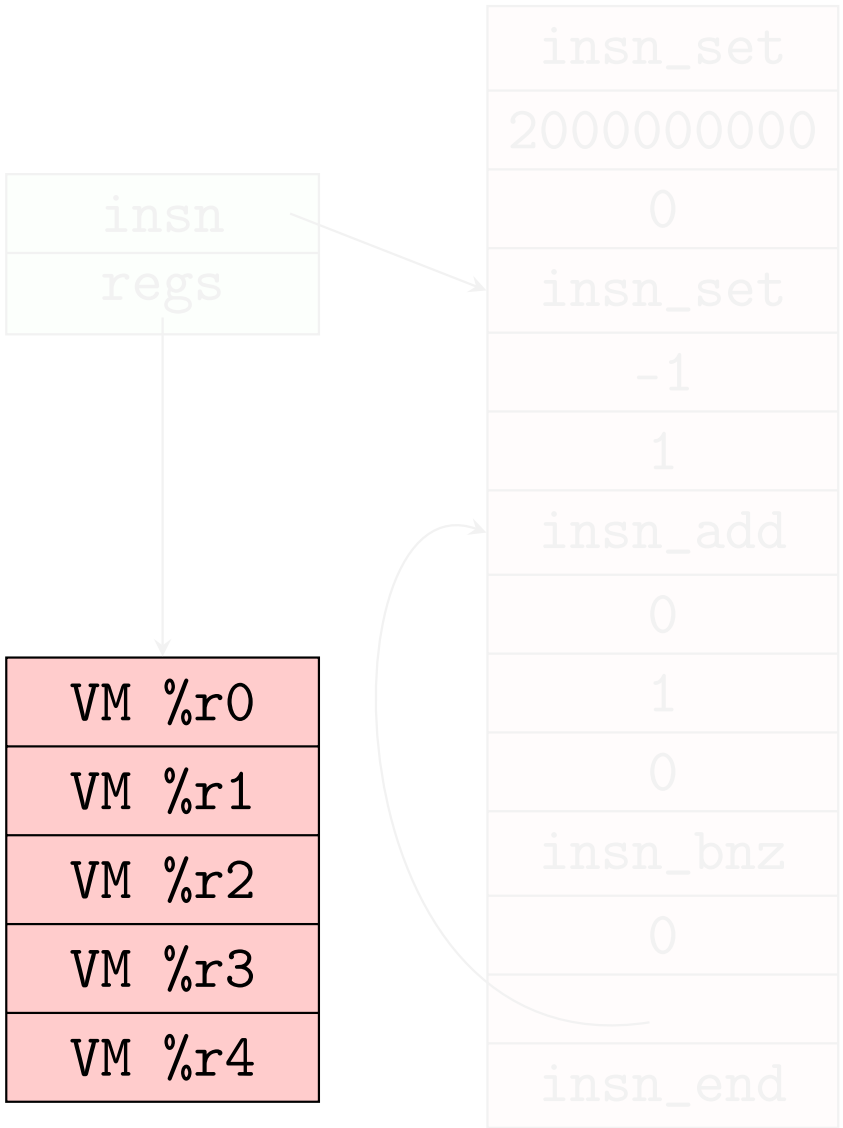
```
      set 2000000000, %r0
      set -1, %r1
$L1:  add %r0, %r1, %r0
      bnz %r0, $L1
      end
```

- VM registers are an array in hardware memory.
- The VM program is an array in hardware memory.
- Only *the interpreter's automatic C variables* are in hardware registers.

The down-counter as a linear program to be interpreted

```
      set 2000000000, %r0
      set -1, %r1
$L1:  add %r0, %r1, %r0
      bnz %r0, $L1
      end
```

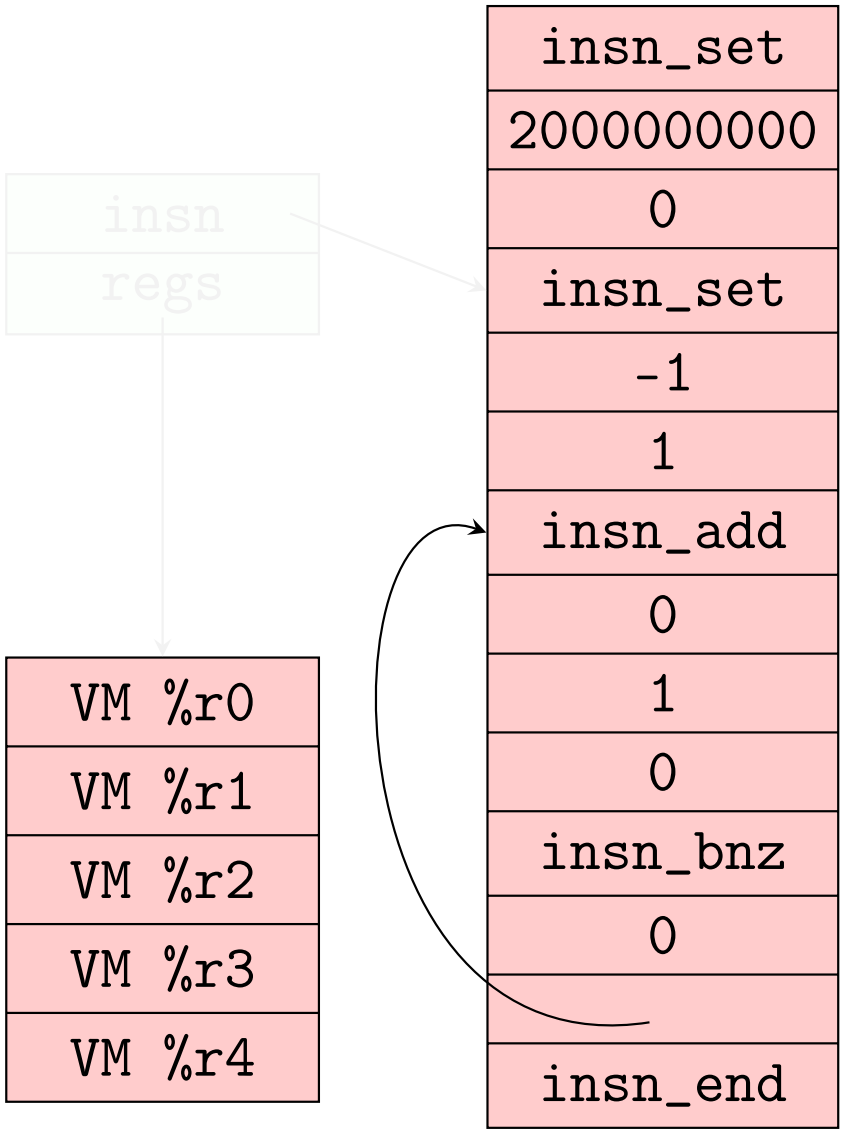
- VM registers are an array in hardware memory.
- The VM program is an array in hardware memory.
- Only *the interpreter's automatic C variables* are in hardware registers.



The down-counter as a linear program to be interpreted

```
      set 2000000000, %r0
      set -1, %r1
$L1:  add %r0, %r1, %r0
      bnz %r0, $L1
      end
```

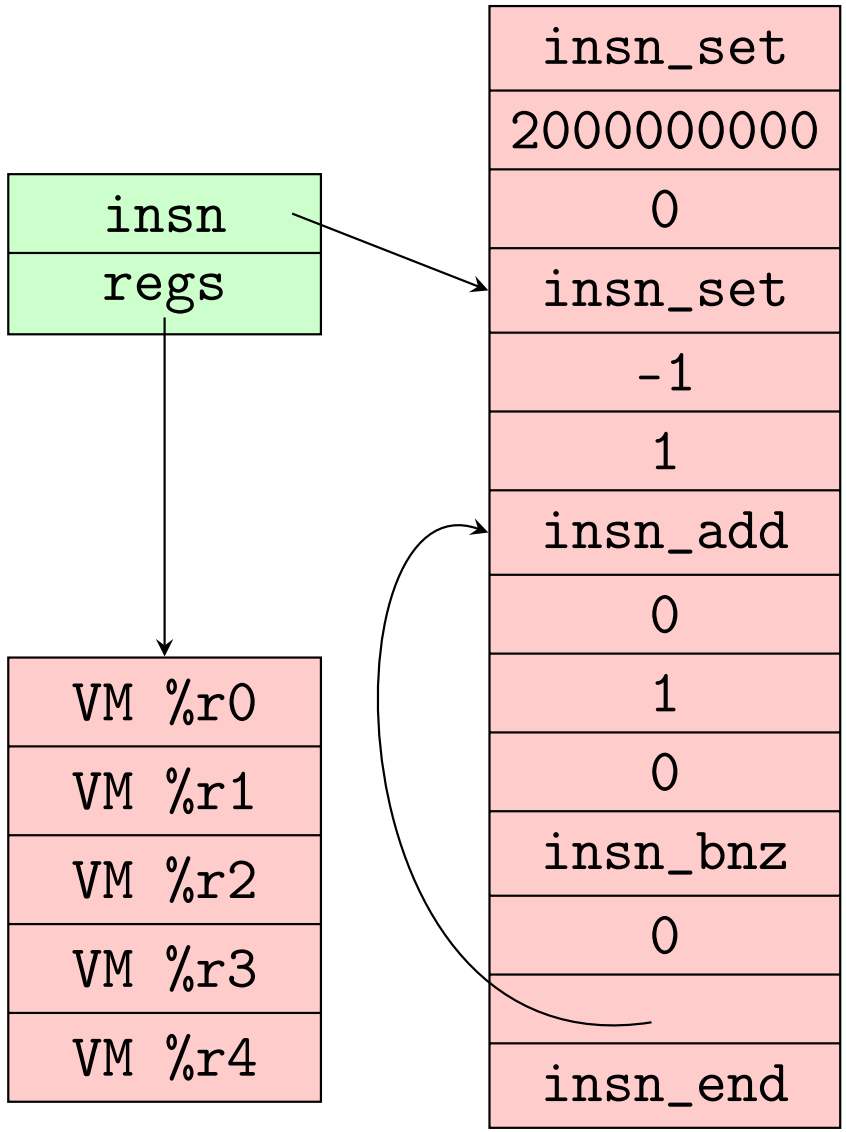
- VM registers are an array in hardware memory.
- The VM program is an array in hardware memory.
- Only *the interpreter's automatic C variables* are in hardware registers.



The down-counter as a linear program to be interpreted

```
      set 2000000000, %r0
      set -1, %r1
$L1:  add %r0, %r1, %r0
      bnz %r0, $L1
      end
```

- VM registers are an array in hardware memory.
- The VM program is an array in hardware memory.
- Only *the interpreter's automatic C variables* are in hardware registers.



The simplest linear-program interpreter

What's the C type of `insn_set`, `insn_add`, `insn_bnz`, `insn_end`?

- It's an `enum insn`: essentially an integer.
- There are also pointers *in* the VM program array from an element to another...
- Linear-program interpreters work best with word-sized data: objects as wide as a hardware register. `union`s are useful for this:

C

```
union value
{
  enum insn in;
  long i; // or another integer type of the right width
  union value *p;
};
```

This interpretation style is called `switch` dispatching.

[switch dispatching: C source and demo]

The simplest linear-program interpreter

What's the C type of `insn_set`, `insn_add`, `insn_bnz`, `insn_end`?

- It's an `enum insn`: essentially an integer.
- There are also pointers *in* the VM program array from an element to another...
- Linear-program interpreters work best with word-sized data: objects as wide as a hardware register. `union`s are useful for this:

```c
union value
{
  enum insn in;
  long i; // or another integer type of the right width
  union value *p;
};
```

This interpretation style is called `switch` dispatching.

[switch dispatching: C source and demo]

The simplest linear-program interpreter

What's the C type of `insn_set`, `insn_add`, `insn_bnz`, `insn_end`?

- It's an `enum insn`: essentially an integer.
- There are also pointers *in* the VM program array from an element to another...
- Linear-program interpreters work best with word-sized data: objects as wide as a hardware register. `union`s are useful for this:

```c
union value
{
  enum insn in;
  long i; // or another integer type of the right width
  union value *p;
};
```

This interpretation style is called `switch` dispatching.

[switch dispatching: C source and demo]

The simplest linear-program interpreter

What's the C type of `insn_set`, `insn_add`, `insn_bnz`, `insn_end`?

- It's an `enum insn`: essentially an integer.
- There are also pointers *in* the VM program array from an element to another...
- Linear-program interpreters work best with word-sized data: objects as wide as a hardware register. `union`s are useful for this:

C

```
union value
{
  enum insn in;
  long i; // or another integer type of the right width
  union value *p;
};
```

This interpretation style is called `switch` dispatching.

[switch dispatching: C source and demo]

The simplest linear-program interpreter

What's the C type of `insn_set`, `insn_add`, `insn_bnz`, `insn_end`?

- It's an `enum insn`: essentially an integer.
- There are also pointers *in* the VM program array from an element to another...
- Linear-program interpreters work best with word-sized data: objects as wide as a hardware register. `union`s are useful for this:

C

```
union value
{
  enum insn in;
  long i; // or another integer type of the right width
  union value *p;
};
```

This interpretation style is called `switch` dispatching.

[switch dispatching: C source and demo]

Problems with `switch`-dispatching

Performance of a `switch`-dispatching interpreter:

- `switch` is somewhat inefficient (range checking)
- The CPU branch target predictor can't work well: one jumping instruction with many possible targets, complex repetition patterns.
- Every VM instruction ends with another jump to the one shared `switch`.

Problems with `switch`-dispatching

Performance of a `switch`-dispatching interpreter:

- `switch` is somewhat inefficient (range checking)
- The CPU branch target predictor can't work well: one jumping instruction with many possible targets, complex repetition patterns.
- Every VM instruction ends with another jump to the one shared `switch`.

Problems with `switch`-dispatching

Performance of a `switch`-dispatching interpreter:

- `switch` is somewhat inefficient (range checking)
- The CPU branch target predictor can't work well: one jumping instruction with many possible targets, complex repetition patterns.
- Every VM instruction ends with another jump to the one shared `switch`.

Problems with `switch`-dispatching

Performance of a `switch`-dispatching interpreter:

- `switch` is somewhat inefficient (range checking)
- The CPU branch target predictor can't work well: one jumping instruction with many possible targets, complex repetition patterns.
- Every VM instruction ends with another jump to the one shared `switch`.

Problems with `switch`-dispatching

Performance of a `switch`-dispatching interpreter:

- `switch` is somewhat inefficient (range checking)
- The CPU branch target predictor can't work well: one jumping instruction with many possible targets, complex repetition patterns.
- Every VM instruction ends with another jump to the one shared `switch`.

Computed goto

GCC introduced the C extension called computed goto or labels-as-values:

- The expression `&& label`, of type `void *`, evaluates to the address of the hardware machine instruction where the labeled code begins; you can store the address and jump to it later.
- The statement `goto *expr` jumps to the result of the evaluation of *expr*.

We can use pointers to native code instead of instead of `enum`s in the VM program, at the beginning of every VM instruction.
This is called direct-threaded code (*nothing to do* with multi-threading).

Computed goto

GCC introduced the C extension called computed goto or labels-as-values:

- The expression `&& label`, of type `void *`, evaluates to the address of the hardware machine instruction where the labeled code begins; you can store the address and jump to it later.
- The statement `goto *expr` jumps to the result of the evaluation of *expr*.

We can use pointers to native code instead of instead of `enum`s in the VM program, at the beginning of every VM instruction.
This is called direct-threaded code (*nothing to do* with multi-threading).

Computed goto

GCC introduced the C extension called computed goto or labels-as-values:

- The expression `&& label`, of type `void *`, evaluates to the address of the hardware machine instruction where the labeled code begins; you can store the address and jump to it later.
- The statement `goto *expr` jumps to the result of the evaluation of *expr*.

We can use pointers to native code instead of instead of enums in the VM program, at the beginning of every VM instruction. This is called direct-threaded code (*nothing to do* with multi-threading).

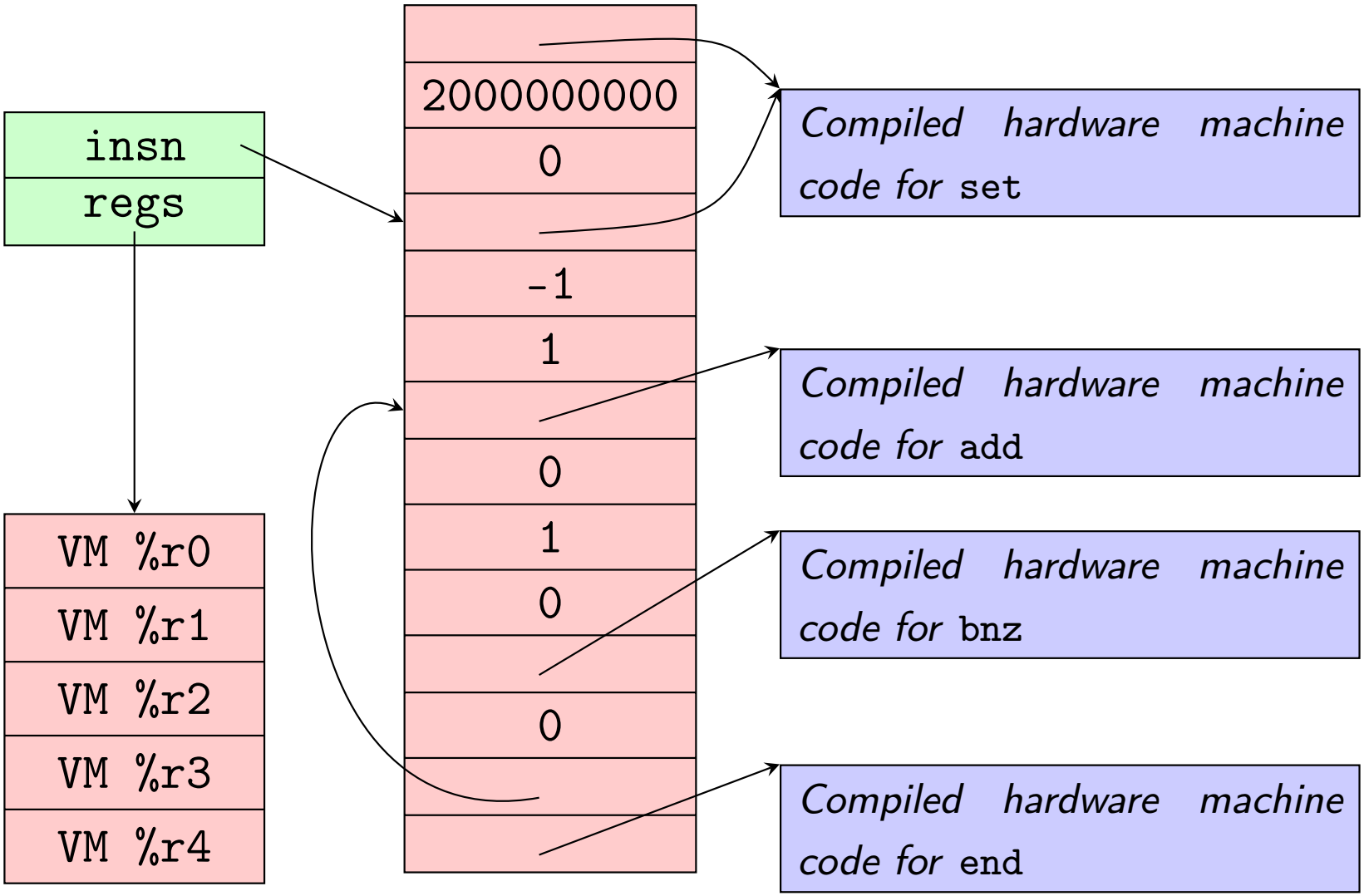
Computed goto

GCC introduced the C extension called computed goto or labels-as-values:

- The expression `&& label`, of type `void *`, evaluates to the address of the hardware machine instruction where the labeled code begins; you can store the address and jump to it later.
- The statement `goto *expr` jumps to the result of the evaluation of *expr*.

We can use pointers to native code instead of instead of `enums` in the VM program, at the beginning of every VM instruction.
This is called direct-threaded code (*nothing to do* with multi-threading).

The down-counter program for a direct-threaded VM



Instead of an `enum` identifier each VM instruction in the VM program begins with a pointer to its native code.

Direct-threaded interpretation

In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded interpretation

In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded interpretation

In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded interpretation

In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded interpretation

In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded interpretation

In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded interpretation

In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded interpretation

In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded interpretation

In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded interpretation

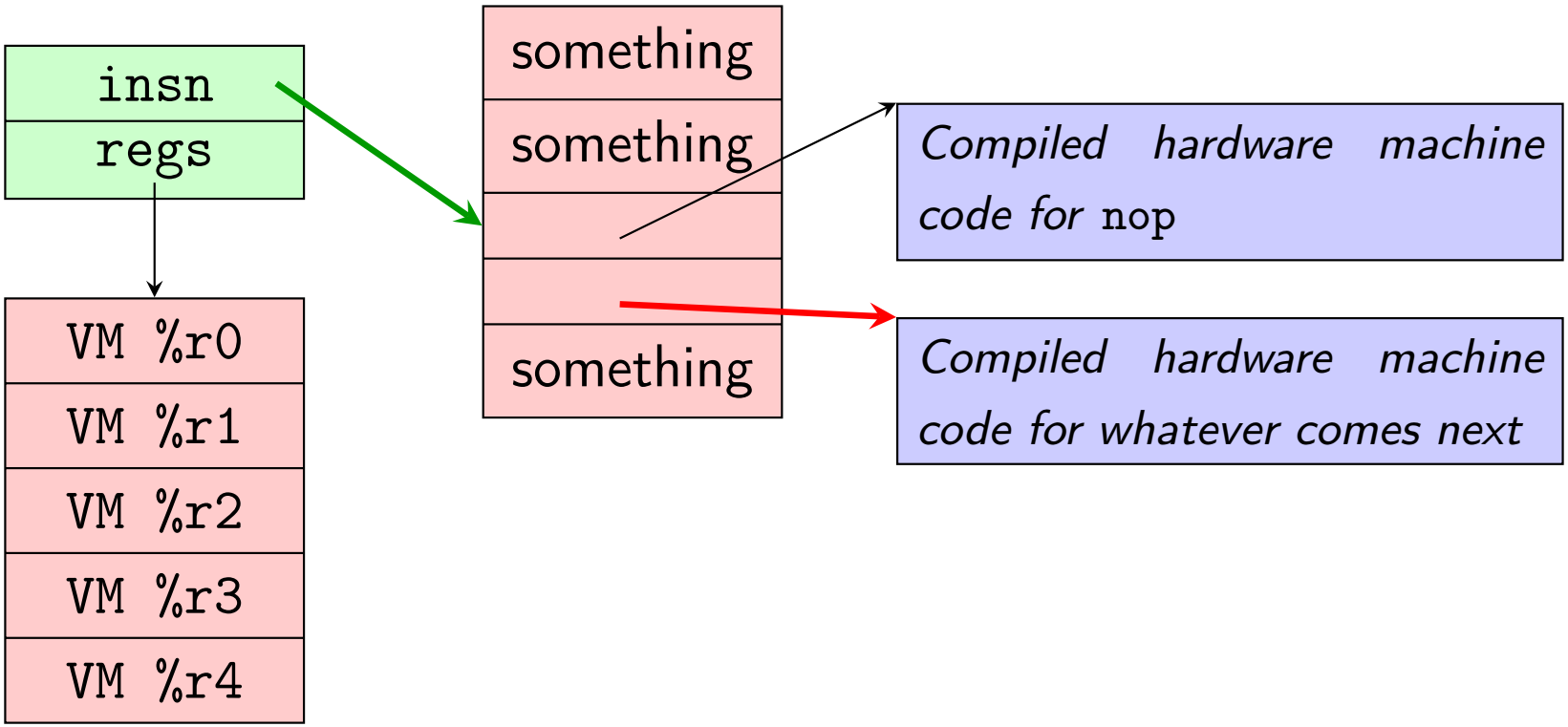
In direct threading:

- interpreting the VM instruction pointed by a C pointer `p` is trivial: `goto *p;`
- there's no `switch`
- no infinite loop or jump to a shared conditional: each VM instruction "falls thru" to the next:
  - move `insn` forward
  - load the next VM instruction code pointer from it
  - `goto *` to the code pointer
    - Many different jumping hardware instructions: less bad for the hardware branch target predictor
- (also, still as *compact in memory* as `switch`-dispatching: useful for small embedded systems, but not particularly for GNU)

[C source and demo]

Direct-threaded fallthru (nop): diagram

The zero-argument VM instruction `nop` does nothing and just falls thru to the next instruction.
The jump destination address is pointed from memory (red arrow).
The green arrow is the pointer `insn`, already in a hardware register.



There is nothing between the code pointer for `nop` and the code pointer for the next VM instruction because `nop` has no arguments.

Direct-threaded fallthru (nop): code

Here's the source for the VM instruction nop in the direct-threading interpreter:

GNU C

```
label_nop:
  insn ++;   // No args to skip, just the code pointer
  goto * insn->label;
```

compiled (x86_64)

```
movq 8(%rax), %rdx #insn is in %rax; load (insn + 1)->label
addq $8, %rax      #advance insn to the next instruction
jmpq *%rdx         #jump to the address we loaded before
```

GCC has put insn in the hardware register %rax. The load (movq on x86_64) follows the red arrow, from %rax + 8. The hardware register %rdx is a temporary, holding the address where to jump.

Direct-threaded fallthru (nop): code

Here's the source for the VM instruction nop in the direct-threading interpreter:

GNU C

```
label_nop:
  insn ++;   // No args to skip, just the code pointer
  goto * insn->label;
```

compiled (x86_64)

```
movq 8(%rax), %rdx #insn is in %rax; load (insn + 1)->label
addq $8, %rax      #advance insn to the next instruction
jmpq *%rdx         #jump to the address we loaded before
```

GCC has put `insn` in the hardware register `%rax`. The load (`movq` on `x86_64`) follows the red arrow, from `%rax + 8`. The hardware register `%rdx` is a temporary, holding the address where to jump.

Direct-threaded fallthru (nop): code

Here's the source for the VM instruction nop in the direct-threading interpreter:

GNU C

```
label_nop:
  insn ++;   // No args to skip, just the code pointer
  goto * insn->label;
```

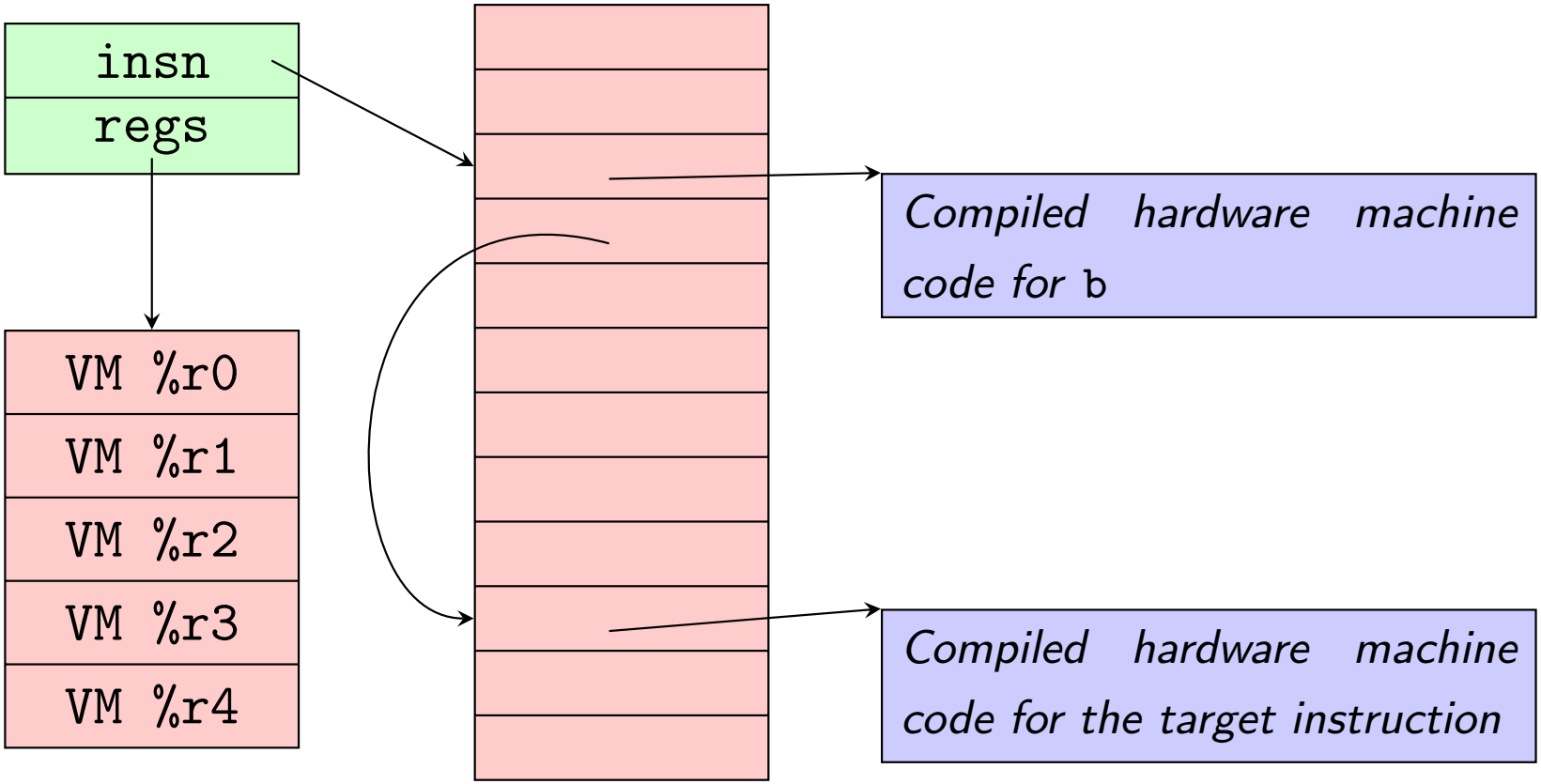
compiled (x86_64)

```
movq 8(%rax), %rdx #insn is in %rax; load (insn + 1)->label
addq $8, %rax      #advance insn to the next instruction
jmpq *%rdx         #jump to the address we loaded before
```

GCC has put insn in the hardware register %rax. The load (movq on x86_64) follows the red arrow, from %rax + 8. The hardware register %rdx is a temporary, holding the address where to jump.

Direct-threaded unconditional branch (b): diagram

The b VM instruction takes a label as its parameter: the next VM program slot after b's code pointer points to the beginning of the target instruction (another slot in the program containing a code pointer).



Direct-threaded unconditional branch (b): code

The (one-argument) VM instruction `b` in the direct-threading interpreter:

GNU C

```
label_b:
  insn = insn[1].p;
  goto * insn->label;
```

compiled (x86_64)

```
movq 8(%rax), %rax # load jump destination from *(insn + 1)
jmpq *(%rax)       # jump indirect via memory: another load
```

The first instruction loads the next `insn`, still pointing within the program array. The jump-via-memory instruction chases a pointer from it and obtains a pointer into a "blue" box, the hardware instruction where to jump where the target VM instruction begins.

Direct-threaded unconditional branch (b): code

The (one-argument) VM instruction `b` in the direct-threading interpreter:

GNU C

```
label_b:
  insn = insn[1].p;
  goto * insn->label;
```

compiled (x86_64)

```
movq 8(%rax), %rax # load jump destination from *(insn + 1)
jmpq *(%rax)       # jump indirect via memory: another load
```

The first instruction loads the next `insn`, still pointing within the program array. The jump-via-memory instruction chases a pointer from it and obtains a pointer into a "blue" box, the hardware instruction where to jump where the target VM instruction begins.

Direct-threaded conditional branch (bnz)

The two-argument VM instruction `bnz` in the direct-threading interpreter:

GNU C

```
label_bnz:
  if (regs[insn[1].i] != 0)
    insn = insn[2].p;
  else
    insn += 3;
  goto * insn->label;
```

compiled (x86_64, simplified)

```
    movq 8(%rax), %rdx
    cmpq $0, -256(%rbp,%rdx,8)
    je   L
    movq 16(%rax), %rax # Like b
    jmpq *(%rax)
L:  addq $24, %rax      # Fallthru
    jmpq *(%rax)
```

Check the condition; if false skip past (`je`) unconditional branch code, and into fallthru dispatch code.

Lots of hardware branches, depending on memory and on each other.

Direct-threaded conditional branch (bnz)

The two-argument VM instruction `bnz` in the direct-threading interpreter:

GNU C

```
label_bnz:
  if (regs[insn[1].i] != 0)
    insn = insn[2].p;
  else
    insn += 3;
  goto * insn->label;
```

compiled (x86_64, simplified)

```
    movq 8(%rax), %rdx
    cmpq $0, -256(%rbp,%rdx,8)
    je   L
    movq 16(%rax), %rax # Like b
    jmpq *(%rax)
L:  addq $24, %rax      # Fallthru
    jmpq *(%rax)
```

Check the condition; if false skip past (je) unconditional branch code, and into fallthru dispatch code.

Lots of hardware branches, depending on memory and on each other.

Direct-threaded conditional branch (bnz)

The two-argument VM instruction `bnz` in the direct-threading interpreter:

GNU C

```
label_bnz:
  if (regs[insn[1].i] != 0)
    insn = insn[2].p;
  else
    insn += 3;
  goto * insn->label;
```

compiled (x86_64, simplified)

```
    movq 8(%rax), %rdx
    cmpq $0, -256(%rbp,%rdx,8)
    je   L
    movq 16(%rax), %rax # Like b
    jmpq *(%rax)
L:  addq $24, %rax      # Fallthru
    jmpq *(%rax)
```

Check the condition; if false skip past (`je`) unconditional branch code, and into fallthru dispatch code.

Lots of hardware branches, depending on memory and on each other.

Direct-threaded conditional branch (bnz)

The two-argument VM instruction `bnz` in the direct-threading interpreter:

GNU C

```
label_bnz:
  if (regs[insn[1].i] != 0)
    insn = insn[2].p;
  else
    insn += 3;
  goto * insn->label;
```

compiled (x86_64, simplified)

```
    movq 8(%rax), %rdx
    cmpq $0, -256(%rbp,%rdx,8)
    je   L
    movq 16(%rax), %rax # Like b
    jmpq *(%rax)
L:  addq $24, %rax      # Fallthru
    jmpq *(%rax)
```

Check the condition; if false skip past (`je`) unconditional branch code, and into fallthru dispatch code.

Lots of hardware branches, depending on memory and on each other.

Direct threading dispatch performance

slow?

Direct threading dispatch performance

The real question is whether we can do better, and where the bottleneck is.

Is branching/fallthru the only source of inefficiency?

[Demo: quick timing against `switch`-dispatching]

Direct threading dispatch performance

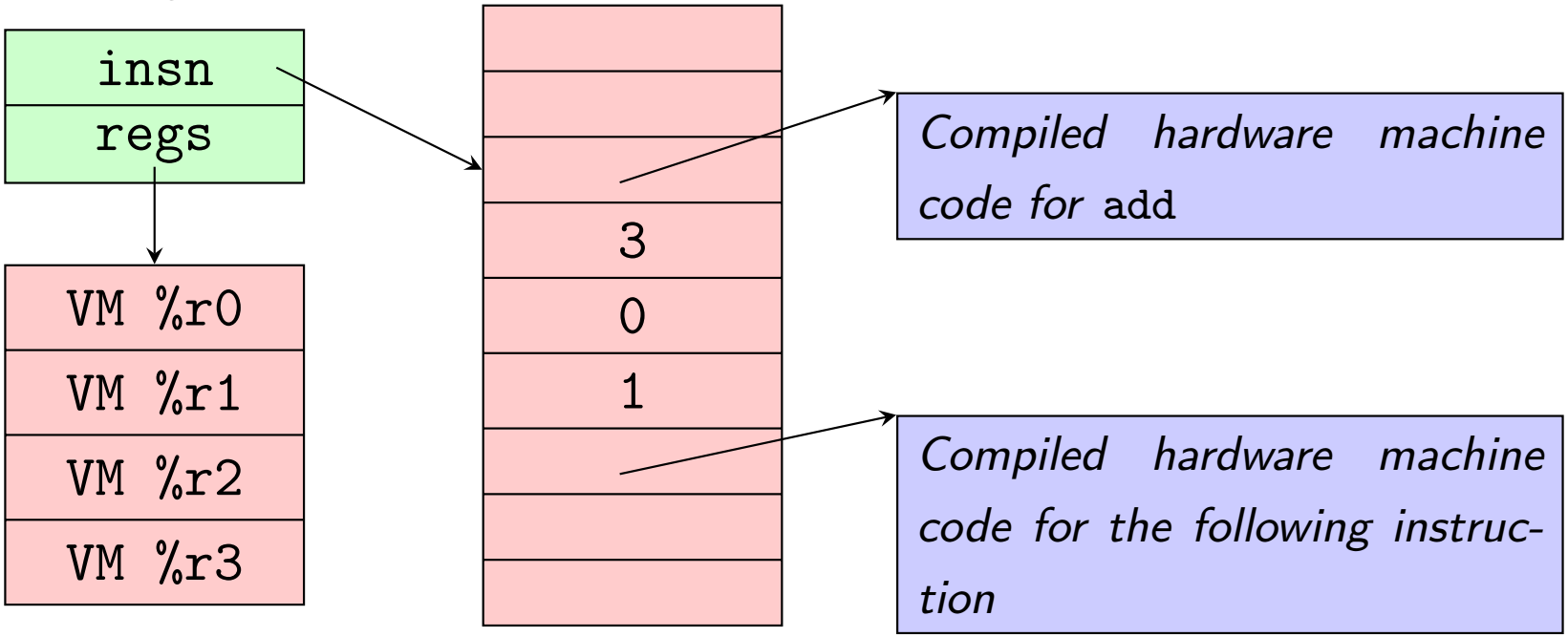
The real question is whether we can do better, and where the bottleneck is.

Is branching/fallthru the only source of inefficiency?

[Demo: quick timing against `switch`*-dispatching]*

(Direct-threaded) VM add: "fundamental"/RISC operations

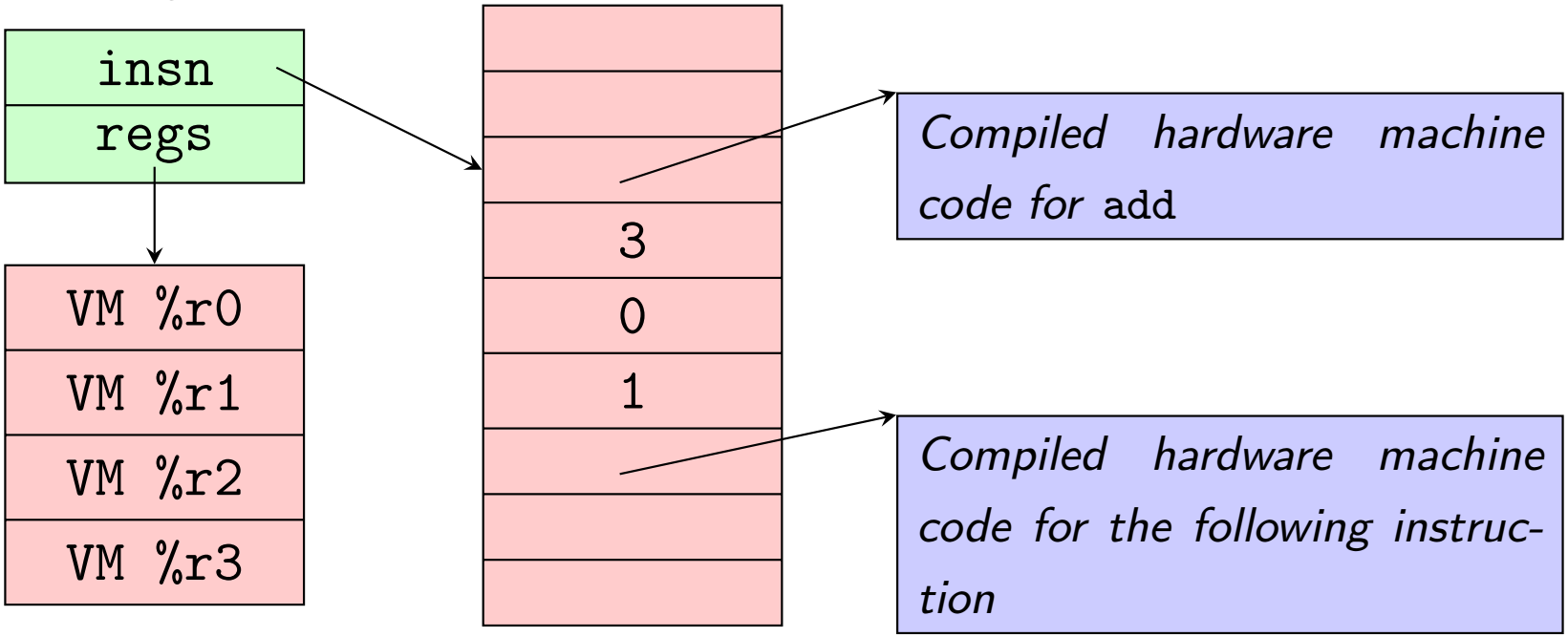
Let's look at how the VM instruction `add %r3, %r0, %r1` is represented in the VM program and what it needs to do in terms of hardware "operations":



- read VM register indices (load from `insn[`k`]` obtaining 3, 0, 1)
- read VM input register contents from the VM register array using input indices (load VM register elements %r3, %r0 using indices 3, 0)
- do the actual sum
- write result into VM register array (store into VM %r1 using index 1)
- fallthru: *increment-load-jump*, as always

(Direct-threaded) VM add: "fundamental"/RISC operations

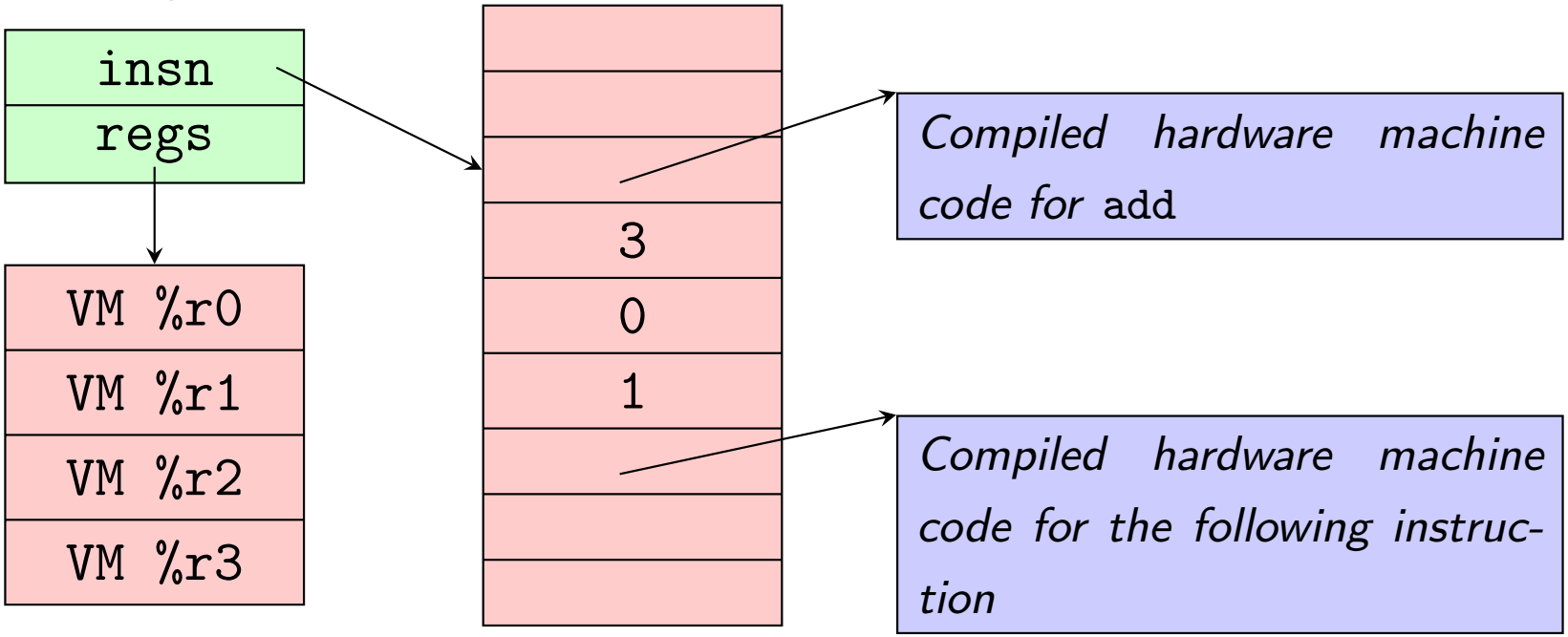
Let's look at how the VM instruction `add %r3, %r0, %r1` is represented in the VM program and what it needs to do in terms of hardware "operations":



- read VM register indices (load from `insn[`k`]` obtaining 3, 0, 1)
- read VM input register contents from the VM register array using input indices (load VM register elements %r3, %r0 using indices 3, 0)
- do the actual sum
- write result into VM register array (store into VM %r1 using index 1)
- fallthru: *increment-load-jump*, as always

(Direct-threaded) VM add: "fundamental"/RISC operations

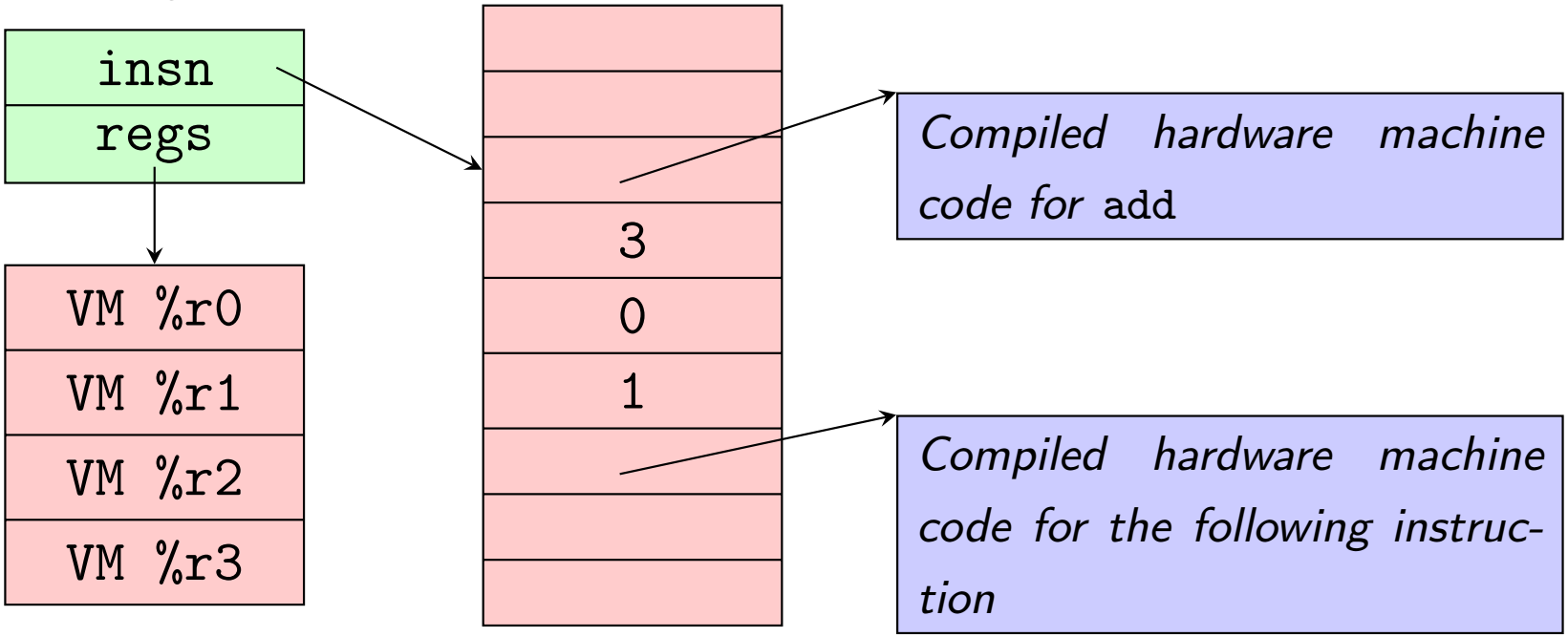
Let's look at how the VM instruction `add %r3, %r0, %r1` is represented in the VM program and what it needs to do in terms of hardware "operations":



- read VM register indices (load from `insn[`k`]` obtaining 3, 0, 1)
- read VM input register contents from the VM register array using input indices (load VM register elements %r3, %r0 using indices 3, 0)
- do the actual sum
- write result into VM register array (store into VM %r1 using index 1)
- fallthru: *increment-load-jump*, as always

(Direct-threaded) VM add: "fundamental"/RISC operations

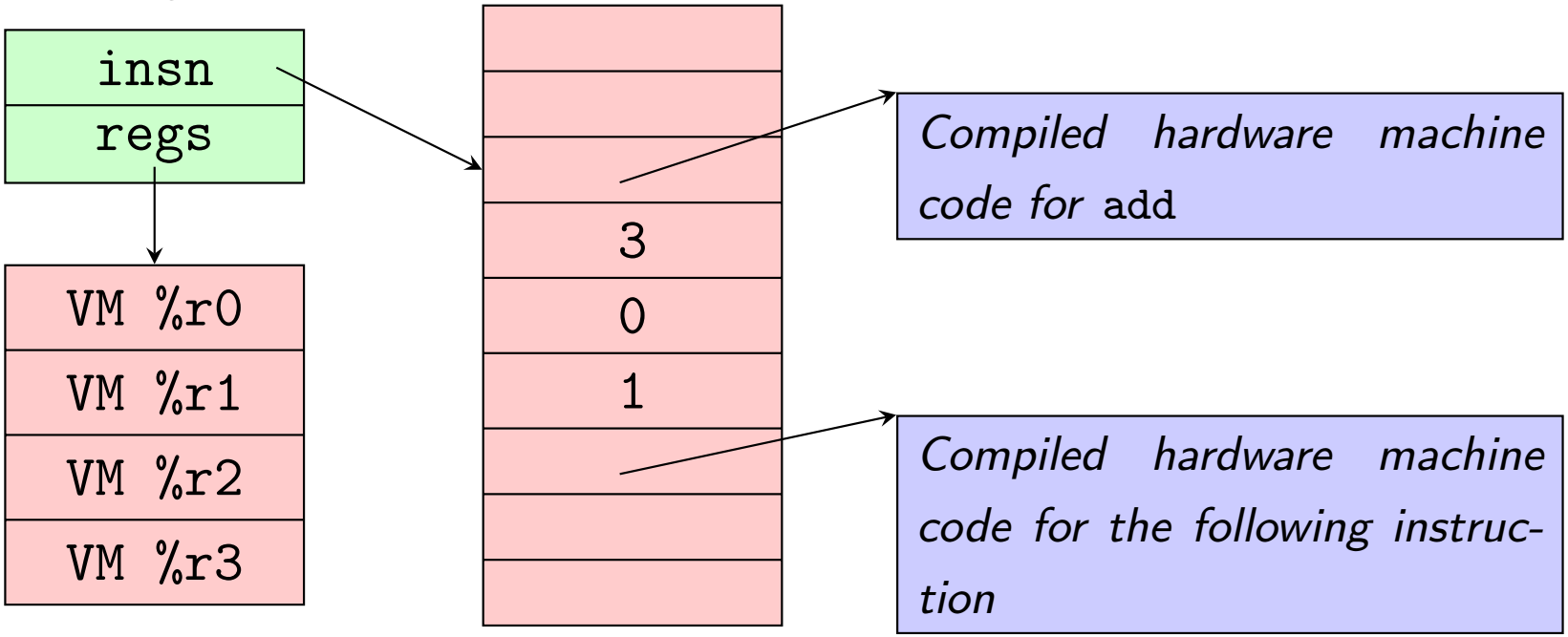
Let's look at how the VM instruction `add %r3, %r0, %r1` is represented in the VM program and what it needs to do in terms of hardware "operations":



- read VM register indices (load from `insn[`k`]` obtaining 3, 0, 1)
- read VM input register contents from the VM register array using input indices (load VM register elements %r3, %r0 using indices 3, 0)
- do the actual sum
- write result into VM register array (store into VM %r1 using index 1)
- fallthru: *increment-load-jump*, as always

(Direct-threaded) VM add: "fundamental"/RISC operations

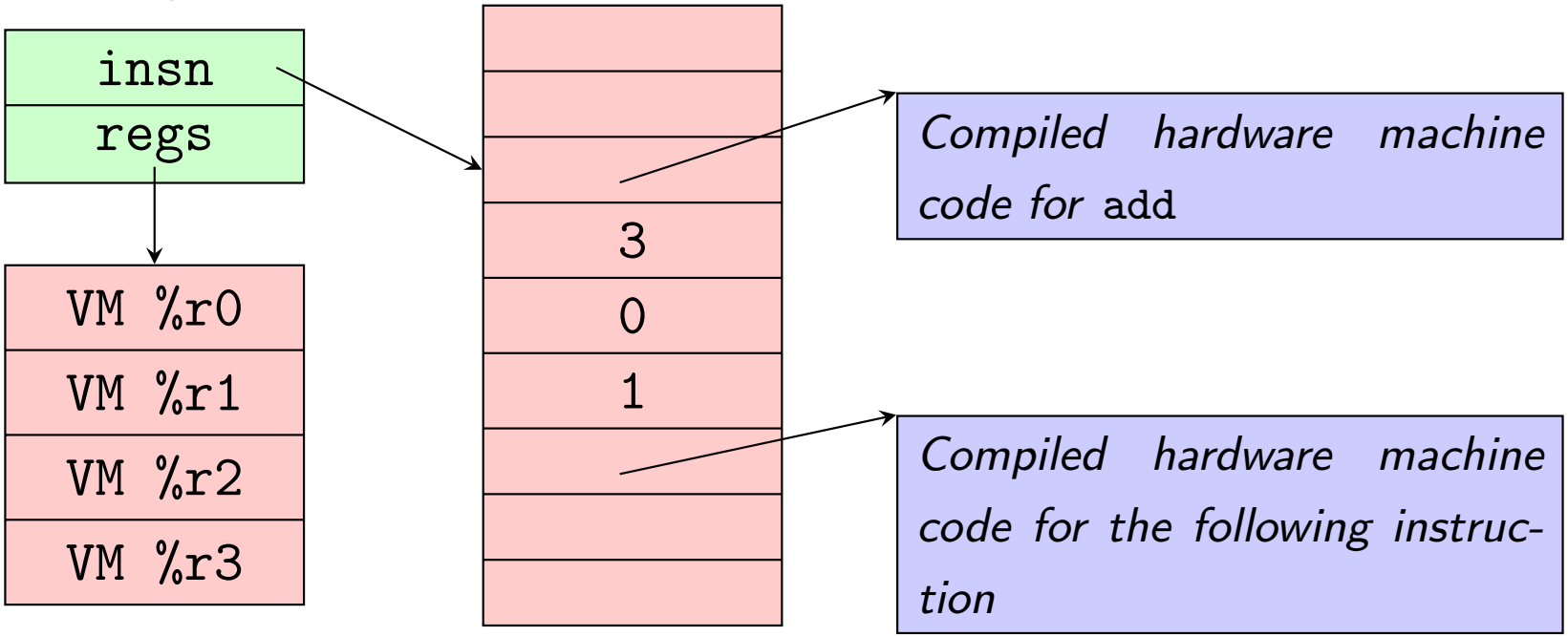
Let's look at how the VM instruction `add %r3, %r0, %r1` is represented in the VM program and what it needs to do in terms of hardware "operations":



- read VM register indices (load from `insn[`k`]` obtaining 3, 0, 1)
- read VM input register contents from the VM register array using input indices (load VM register elements %r3, %r0 using indices 3, 0)
- do the actual sum
- write result into VM register array (store into VM %r1 using index 1)
- fallthru: *increment-load-jump*, as always

(Direct-threaded) VM add: "fundamental"/RISC operations

Let's look at how the VM instruction `add %r3, %r0, %r1` is represented in the VM program and what it needs to do in terms of hardware "operations":



- read VM register indices (load from `insn[`k`]` obtaining 3, 0, 1)
- read VM input register contents from the VM register array using input indices (load VM register elements %r3, %r0 using indices 3, 0)
- do the actual sum
- write result into VM register array (store into VM %r1 using index 1)
- fallthru: *increment-load-jump*, as always

The VM instruction add (here direct-threaded), compiled

Is our three-operand add simple and fast, at least on a CISC?

GNU C

```
label_add:
  regs[insn[3].i]
    = (  regs[insn[1].i]
       + regs[insn[2].i]);
  insn += 4;
  goto * insn->label;
```

compiled (x86_64, simplified)

```
movq 8(%rax), %rsi
movq 16(%rax), %rdx
addq $32, %rax
movq -8(%rax), %rcx
movq -256(%rbp,%rdx,8), %rdx
addq -256(%rbp,%rsi,8), %rdx  # +
movq %rdx, -256(%rbp,%rcx,8)
movq (%rax), %rdx
jmpq *rdx
```

- the actual addition costs only one hardware instruction (the second addq, in black [which also includes one memory access]).
- Fallthru to the next VM instruction: three hardware instructions (*increment-load-jump*).
- The other five hardware instructions only serve to access VM registers (and on RISCs it's even worse).

The VM instruction add (here direct-threaded), compiled

Is our three-operand add simple and fast, at least on a CISC?

GNU C

```
label_add:
  regs[insn[3].i]
    = (  regs[insn[1].i]
       + regs[insn[2].i]);
  insn += 4;
  goto * insn->label;
```

compiled (x86_64, simplified)

```
movq 8(%rax), %rsi
movq 16(%rax), %rdx
addq $32, %rax
movq -8(%rax), %rcx
movq -256(%rbp,%rdx,8), %rdx
addq -256(%rbp,%rsi,8), %rdx  # +
movq %rdx, -256(%rbp,%rcx,8)
movq (%rax), %rdx
jmpq *rdx
```

- the actual addition costs only one hardware instruction (the second addq, in black [which also includes one memory access]).
- Fallthru to the next VM instruction: three hardware instructions (*increment-load-jump*).
- The other five hardware instructions only serve to access VM registers (and on RISCs it's even worse).

The VM instruction add (here direct-threaded), compiled

Is our three-operand add simple and fast, at least on a CISC?

GNU C

```
label_add:
  regs[insn[3].i]
    = (  regs[insn[1].i]
       + regs[insn[2].i]);
  insn += 4;
  goto * insn->label;
```

compiled (x86_64, simplified)

```
movq 8(%rax), %rsi
movq 16(%rax), %rdx
addq $32, %rax
movq -8(%rax), %rcx
movq -256(%rbp,%rdx,8), %rdx
addq -256(%rbp,%rsi,8), %rdx  # +
movq %rdx, -256(%rbp,%rcx,8)
movq (%rax), %rdx
jmpq *rdx
```

- the actual addition costs only one hardware instruction (the second addq, in black [which also includes one memory access]).
- Fallthru to the next VM instruction: three hardware instructions (*increment-load-jump*).
- The other five hardware instructions only serve to access VM registers (and on RISCs it's even worse).

The VM instruction add (here direct-threaded), compiled

Is our three-operand add simple and fast, at least on a CISC?

GNU C

```
label_add:
  regs[insn[3].i]
    = (  regs[insn[1].i]
       + regs[insn[2].i]);
  insn += 4;
  goto * insn->label;
```

compiled (x86_64, simplified)

```
movq 8(%rax), %rsi
movq 16(%rax), %rdx
addq $32, %rax
movq -8(%rax), %rcx
movq -256(%rbp,%rdx,8), %rdx
addq -256(%rbp,%rsi,8), %rdx  # +
movq %rdx, -256(%rbp,%rcx,8)
movq (%rax), %rdx
jmpq *rdx
```

- the actual addition costs only one hardware instruction (the second addq, in black [which also includes one memory access]).
- Fallthru to the next VM instruction: three hardware instructions (*increment-load-jump*).
- The other five hardware instructions only serve to access VM registers (and on RISCs it's even worse).

The VM instruction add (here direct-threaded), compiled

Is our three-operand add simple and fast, at least on a CISC?

GNU C

```
label_add:
  regs[insn[3].i]
    = (  regs[insn[1].i]
       + regs[insn[2].i]);
  insn += 4;
  goto * insn->label;
```

compiled (x86_64, simplified)

```
movq 8(%rax), %rsi
movq 16(%rax), %rdx
addq $32, %rax
movq -8(%rax), %rcx
movq -256(%rbp,%rdx,8), %rdx
addq -256(%rbp,%rsi,8), %rdx  # +
movq %rdx, -256(%rbp,%rcx,8)
movq (%rax), %rdx
jmpq *rdx
```

- the actual addition costs only one hardware instruction (the second addq, in black [which also includes one memory access]).
- Fallthru to the next VM instruction: three hardware instructions (*increment-load-jump*).
- The other five hardware instructions only serve to access VM registers (and on RISCs it's even worse).

(Direct-threaded) VM add: register indices and shifts

In the C code for VM instructions we access VM register contents with expressions such as `regs[`*idx*`]`, where *idx* is usually `insn[`*k*`].i` for some constant *k*.

Reading `insn[`*k*`].i` into *idx* costs one load instruction (register plus a known constant offset). Loading `regs[`*idx*`]` is more delicate: the address to load from is

$$\texttt{regs} + idx \cdot w$$

where w is the word size in bytes (4 on 32-bit machines, 8 on 64-bit machines). The multiplication requires a separate shift instruction on most RISC machines [plus possibly yet another instruction for summing `regs` and $(idx \cdot w)$: needed on RISC-V, MIPS, Alpha].

Shifting at run time is silly: instead of keeping VM register indices in the VM program we can keep VM register offsets from `regs`, or in other words we can keep pre-shifted register indices.

(Direct-threaded) VM add: register indices and shifts

In the C code for VM instructions we access VM register contents with expressions such as `regs[`*idx*`]`, where *idx* is usually `insn[`*k*`].i` for some constant *k*.

Reading `insn[`*k*`].i` into *idx* costs one load instruction (register plus a known constant offset). Loading `regs[`*idx*`]` is more delicate: the address to load from is

$$\texttt{regs} + idx \cdot w$$

where w is the word size in bytes (4 on 32-bit machines, 8 on 64-bit machines). The multiplication requires a separate shift instruction on most RISC machines [plus possibly yet another instruction for summing `regs` and $(idx \cdot w)$: needed on RISC-V, MIPS, Alpha].

Shifting at run time is silly: instead of keeping VM register indices in the VM program we can keep VM register offsets from `regs`, or in other words we can keep pre-shifted register indices.

(Direct-threaded) VM add: register indices and shifts

In the C code for VM instructions we access VM register contents with expressions such as `regs[`*idx*`]`, where *idx* is usually `insn[`*k*`].i` for some constant *k*.

Reading `insn[`*k*`].i` into *idx* costs one load instruction (register plus a known constant offset). Loading `regs[`*idx*`]` is more delicate: the address to load from is

$$\texttt{regs} + idx \cdot w$$

where *w* is the word size in bytes (4 on 32-bit machines, 8 on 64-bit machines). The multiplication requires a separate shift instruction on most RISC machines [plus possibly yet another instruction for summing `regs` and $(idx \cdot w)$: needed on RISC-V, MIPS, Alpha].

Shifting at run time is silly: instead of keeping VM register indices in the VM program we can keep VM register offsets from `regs`, or in other words we can keep pre-shifted register indices.

(Direct-threaded) VM add: register indices and shifts

In the C code for VM instructions we access VM register contents with expressions such as `regs[`*idx*`]`, where *idx* is usually `insn[`*k*`].i` for some constant *k*.

Reading `insn[`*k*`].i` into *idx* costs one load instruction (register plus a known constant offset). Loading `regs[`*idx*`]` is more delicate: the address to load from is

$$\texttt{regs} + idx \cdot w$$

where w is the word size in bytes (4 on 32-bit machines, 8 on 64-bit machines). The multiplication requires a separate shift instruction on most RISC machines [plus possibly yet another instruction for summing `regs` and $(idx \cdot w)$: needed on RISC-V, MIPS, Alpha].

Shifting at run time is silly: instead of keeping VM register indices in the VM program we can keep VM register offsets from `regs`, or in other words we can keep pre-shifted register indices.

(Direct-threaded) VM add: operation dependency graph

"$a \to b$" means that a uses the result of b, so b is executed before a. Thick arrows mean high latencies ($\sim 3\tau$).

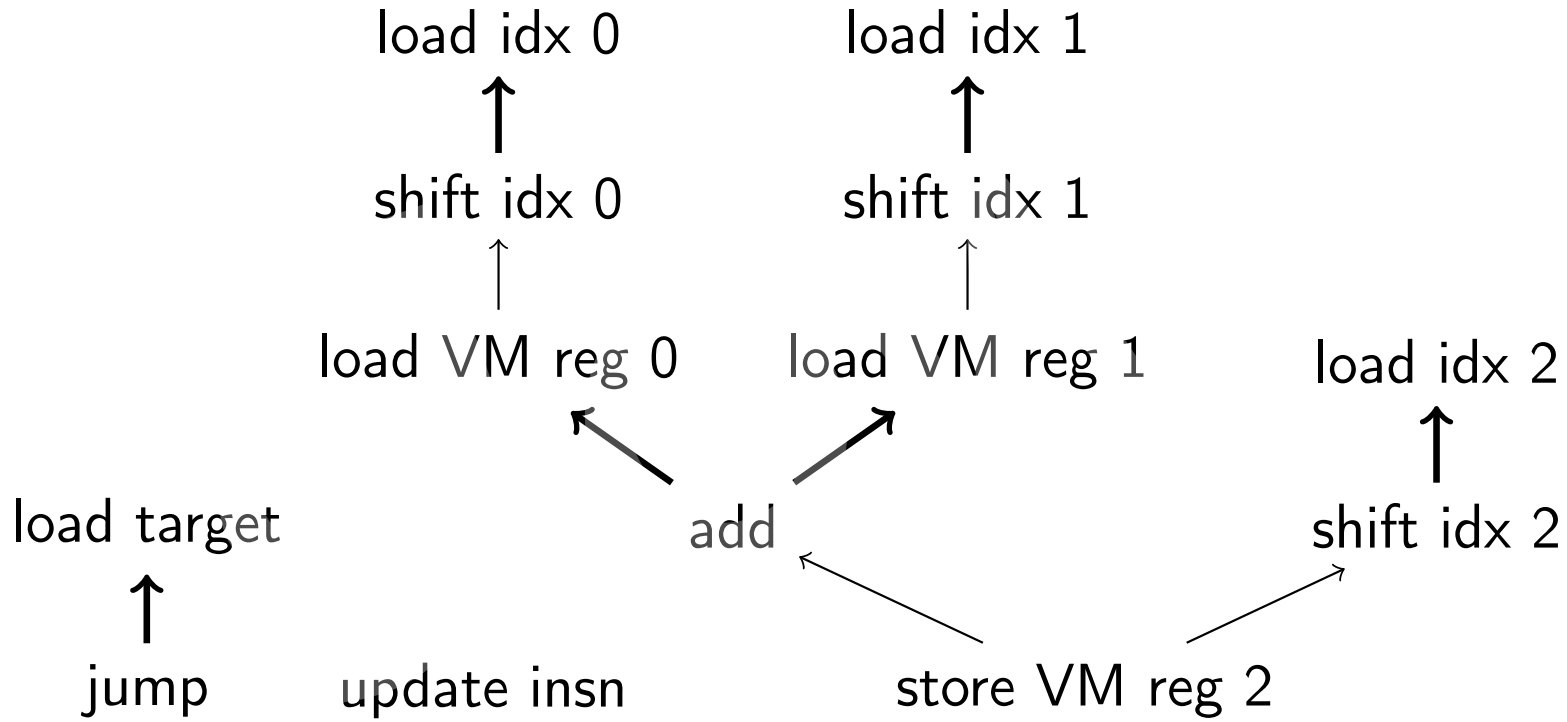
[Register index shifts shown, offset sums to `regs` base *not* shown]

Two long dependency chains, each including two loads: *load*←*shift*←*load*←*add*←*store*. $\sim 6\tau$ latency just from the loads, with ideal Instruction-Level Parallelism! In practice it will be worse.

(Direct-threaded) VM add: operation dependency graph

"$a \to b$" means that a uses the result of b, so b is executed before a. Thick arrows mean high latencies ($\sim 3\tau$).

[Register index shifts shown, offset sums to `regs` base *not* shown]

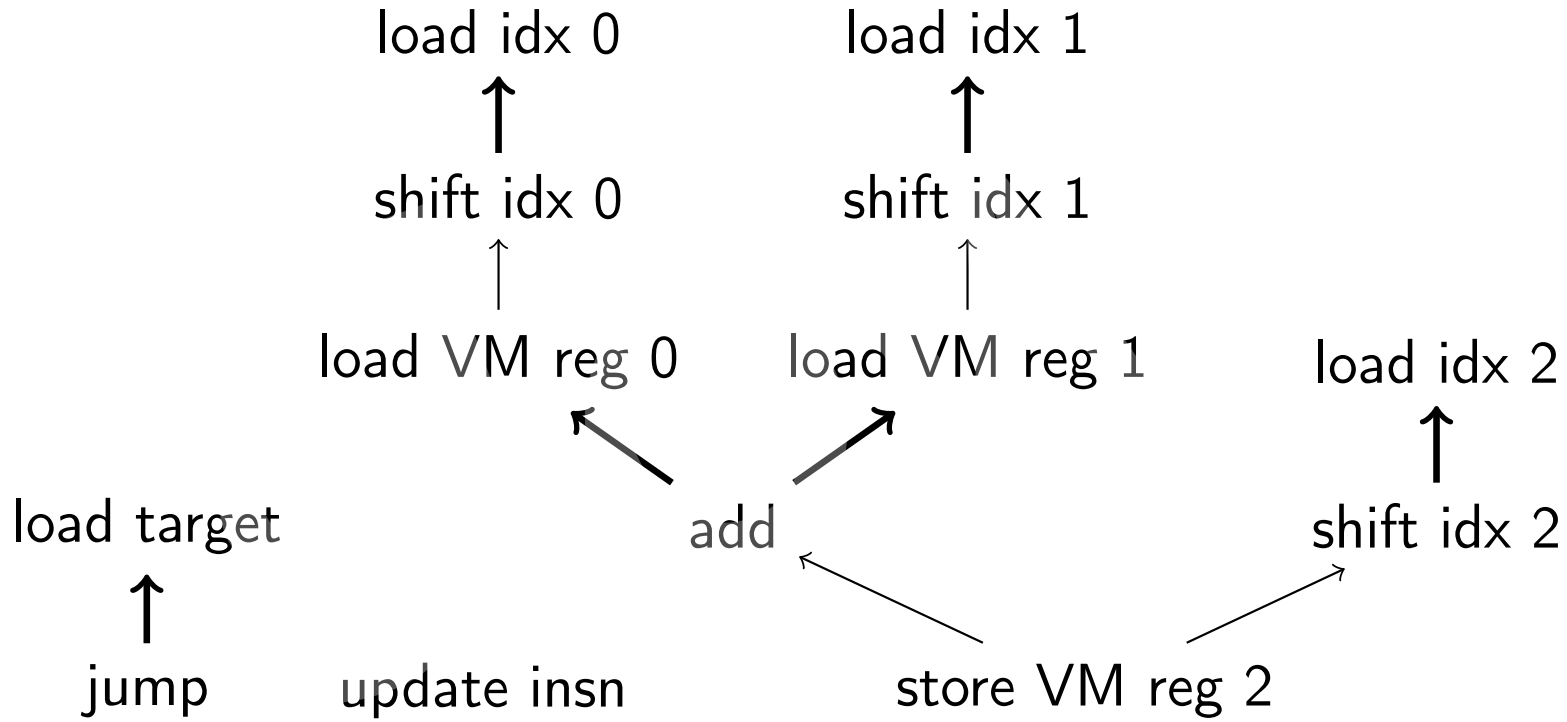


Two long dependency chains, each including two loads: *load←shift←load←add←store*. $\sim 6\tau$ latency just from the loads, with ideal Instruction-Level Parallelism! In practice it will be worse.

(Direct-threaded) VM add: operation dependency graph

"$a \to b$" means that a uses the result of b, so b is executed before a. Thick arrows mean high latencies ($\sim 3\tau$).

[Register index shifts shown, offset sums to `regs` base *not* shown]

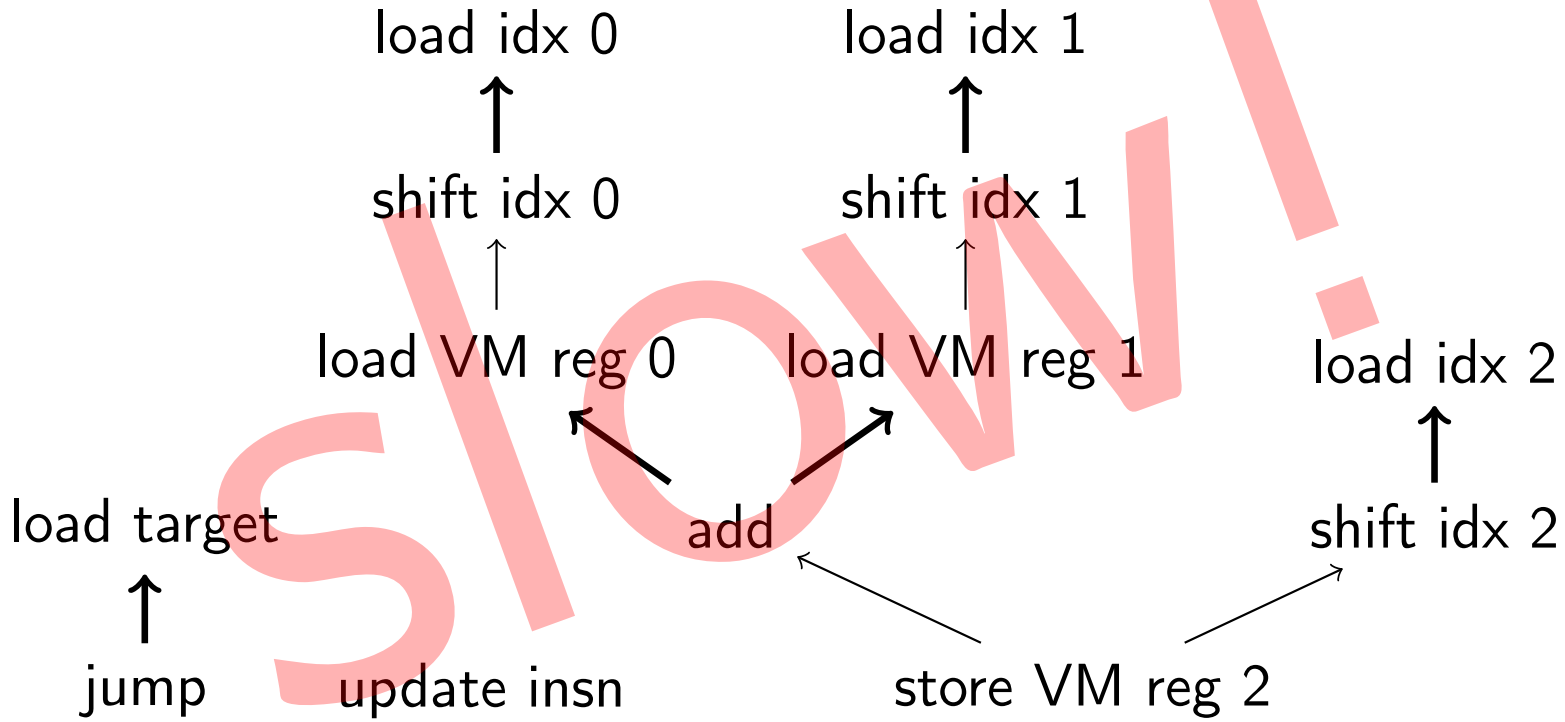


Two long dependency chains, each including two loads: *load*←*shift*←*load*←*add*←*store*. $\sim 6\tau$ latency just from the loads, with ideal Instruction-Level Parallelism! In practice it will be worse.

(Direct-threaded) VM add: operation dependency graph

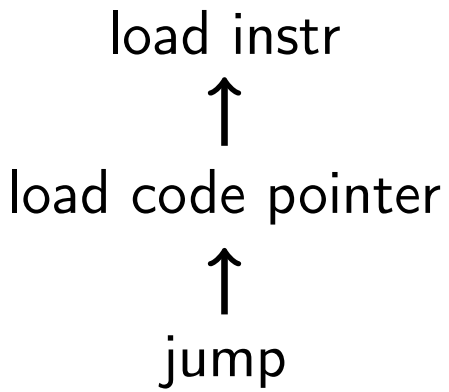
"$a \to b$" means that a uses the result of b, so b is executed before a. Thick arrows mean high latencies ($\sim 3\tau$).

[Register index shifts shown, offset sums to `regs` base *not* shown]



Two long dependency chains, each including two loads: *load*←*shift*←*load*←*add*←*store*. $\sim 6\tau$ latency just from the loads, with ideal Instruction-Level Parallelism! In practice it will be worse.

(Direct-threaded) VM ƀ: operation dependency graph

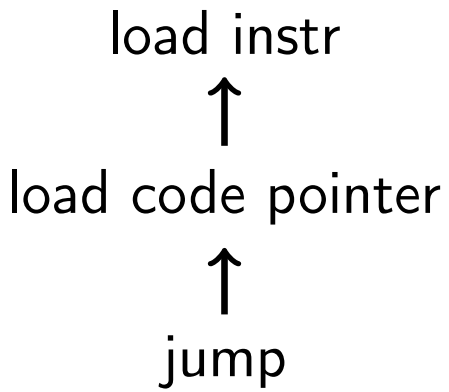


Longest (and only) dependency chain *load*←*load*←*jump*. A VM unconditional branch has latency similar to a VM add; a VM ƀ can easily be faster than a VM add if the hardware branch target predictor does its job.

VMs and hardware machines can have very different performance profiles.

[I've understood, too late to make the change before the GHM, that this is optimizable. Can you see how? *Hint: ƀ can have two arguments instead of one, at least in the memory representation of the program.*]

(Direct-threaded) VM ƀ: operation dependency graph

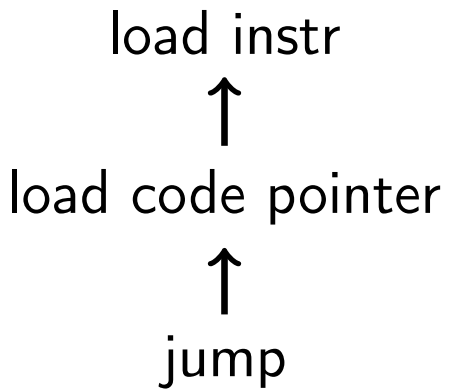


Longest (and only) dependency chain *load*←*load*←*jump*. A VM unconditional branch has latency similar to a VM add; a VM ƀ can easily be faster than a VM add if the hardware branch target predictor does its job.

VMs and hardware machines can have very different performance profiles.

[I've understood, too late to make the change before the GHM, that this is optimizable. Can you see how? *Hint: ƀ can have two arguments instead of one, at least in the memory representation of the program.*]

(Direct-threaded) VM ƀ: operation dependency graph

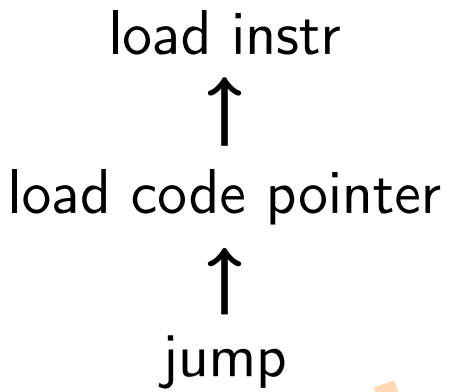


Longest (and only) dependency chain *load*←*load*←*jump*. A VM unconditional branch has latency similar to a VM add; a VM ƀ can easily be faster than a VM add if the hardware branch target predictor does its job.

VMs and hardware machines can have very different performance profiles.

[I've understood, too late to make the change before the GHM, that this is optimizable. Can you see how? *Hint: ƀ can have two arguments instead of one, at least in the memory representation of the program.*]

(Direct-threaded) VM ƀ: operation dependency graph



Longest (and only) dependency chain *load*←*load*←*jump*. A VM unconditional branch has latency similar to a VM add; a VM ƀ can easily be faster than a VM add if the hardware branch target predictor does its job.

VMs and hardware machines can have very different performance profiles.

[I've understood, too late to make the change before the GHM, that this is optimizable. Can you see how? *Hint: ƀ can have two arguments instead of one, at least in the memory representation of the program.*]

What if we used a stack instead of VM registers?

Stack-oriented VM instructions replace the top few elements of a stack with the result of an operation. For example `stack_add` (zero arguments) could pop two elements (say, 5 and 6) from the stack and push their sum (11). This idea is about using stacks instead of VM registers, not just call stacks.

The authors of [Shi et al., 2005], in other works as well, argue from experimental data that direct-threaded register VMs are faster than direct-threaded stack VMs (same model I'm presenting here, stack code machine-translated to VM-register code with optimizations).

Unfortunately it's difficult to replicate their measurements.
I wonder if their results still hold today, with our proportionally slower L1d caches and better branch predictors. [Still, stack code takes more instructions to do the same work, today like in 2005]

This is why I have doubts:

What if we used a stack instead of VM registers?

Stack-oriented VM instructions replace the top few elements of a stack with the result of an operation. For example `stack_add` (zero arguments) could pop two elements (say, 5 and 6) from the stack and push their sum (11). This idea is about using stacks instead of VM registers, not just call stacks.

The authors of [Shi et al., 2005], in other works as well, argue from experimental data that direct-threaded register VMs are faster than direct-threaded stack VMs (same model I'm presenting here, stack code machine-translated to VM-register code with optimizations).

Unfortunately it's difficult to replicate their measurements.
I wonder if their results still hold today, with our proportionally slower L1d caches and better branch predictors. [Still, stack code takes more instructions to do the same work, today like in 2005]

This is why I have doubts:

What if we used a stack instead of VM registers?

Stack-oriented VM instructions replace the top few elements of a stack with the result of an operation. For example `stack_add` (zero arguments) could pop two elements (say, 5 and 6) from the stack and push their sum (11). This idea is about using stacks instead of VM registers, not just call stacks.

The authors of [Shi et al., 2005], in other works as well, argue from experimental data that direct-threaded register VMs are faster than direct-threaded stack VMs (same model I'm presenting here, stack code machine-translated to VM-register code with optimizations).

Unfortunately it's difficult to replicate their measurements.
I wonder if their results still hold today, with our proportionally slower L1d caches and better branch predictors. [Still, stack code takes more instructions to do the same work, today like in 2005]

This is why I have doubts:

What if we used a stack instead of VM registers?

Stack-oriented VM instructions replace the top few elements of a stack with the result of an operation. For example `stack_add` (zero arguments) could pop two elements (say, 5 and 6) from the stack and push their sum (11). This idea is about using stacks instead of VM registers, not just call stacks.

The authors of [Shi et al., 2005], in other works as well, argue from experimental data that direct-threaded register VMs are faster than direct-threaded stack VMs (same model I'm presenting here, stack code machine-translated to VM-register code with optimizations).

Unfortunately it's difficult to replicate their measurements.
I wonder if their results still hold today, with our proportionally slower L1d caches and better branch predictors. [Still, stack code takes more instructions to do the same work, today like in 2005]

This is why I have doubts:

What if we used a stack instead of VM registers?

Stack-oriented VM instructions replace the top few elements of a stack with the result of an operation. For example `stack_add` (zero arguments) could pop two elements (say, 5 and 6) from the stack and push their sum (11). This idea is about using stacks instead of VM registers, not just call stacks.

The authors of [Shi et al., 2005], in other works as well, argue from experimental data that direct-threaded register VMs are faster than direct-threaded stack VMs (same model I'm presenting here, stack code machine-translated to VM-register code with optimizations).

Unfortunately it's difficult to replicate their measurements.
I wonder if their results still hold today, with our proportionally slower L1d caches and better branch predictors. [Still, stack code takes more instructions to do the same work, today like in 2005]

This is why I have doubts:

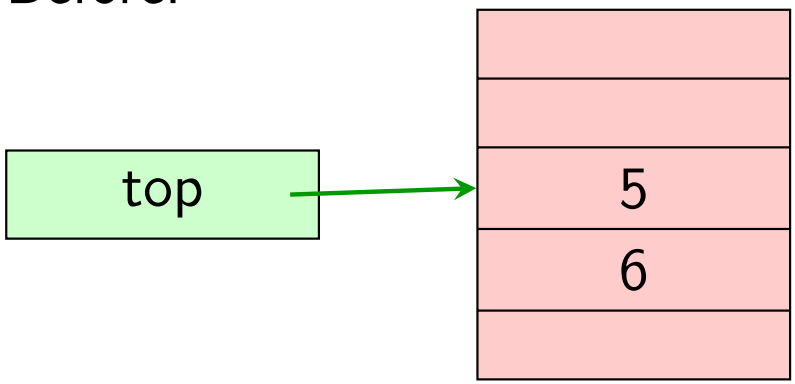
Naïve stack implementation

Suppose the VM has a stack in a hardware memory array, with a top-of-stack *pointer* in a hardware register. This is a zero-argument `stack_add` VM instruction:

GNU C

```
label_stack_add:
  top [-1] = top [-1] + top [0];
  top --;
  /* Fallthru code omitted, same as always. */
```

Before:



After:

Two (independent) loads, one store. This looks better than our VM-register add: constant offsets from top, no index/offset loads.

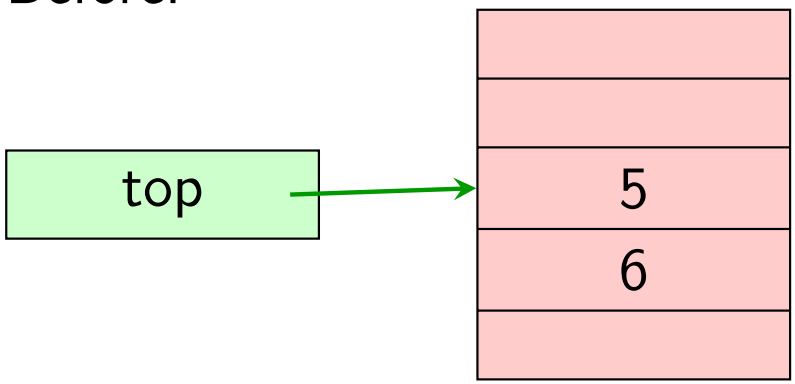
Naïve stack implementation

Suppose the VM has a stack in a hardware memory array, with a top-of-stack *pointer* in a hardware register. This is a zero-argument `stack_add` VM instruction:

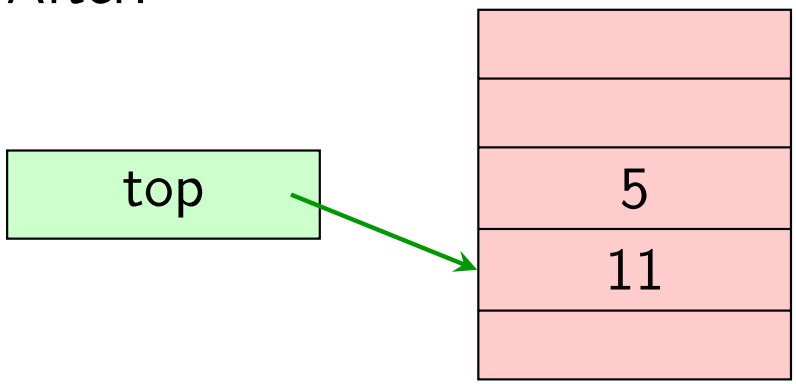
GNU C

```
label_stack_add:
  top [-1] = top [-1] + top [0];
  top --;
  /* Fallthru code omitted, same as always. */
```

Before:



After:



Two (independent) loads, one store. This looks better than our VM-register add: constant offsets from top, no index/offset loads.

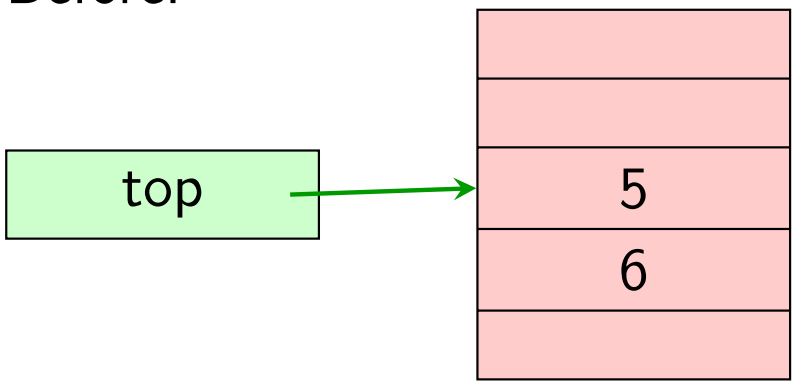
Naïve stack implementation

Suppose the VM has a stack in a hardware memory array, with a top-of-stack *pointer* in a hardware register. This is a zero-argument `stack_add` VM instruction:

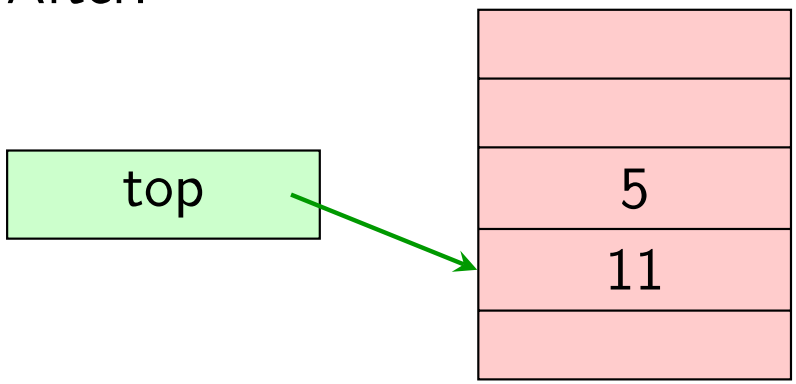
GNU C

```
label_stack_add:
  top [-1] = top [-1] + top [0];
  top --;
  /* Fallthru code omitted, same as always. */
```

Before:



After:



Two (independent) loads, one store. This looks better than our VM-register add: constant offsets from top, no index/offset loads.

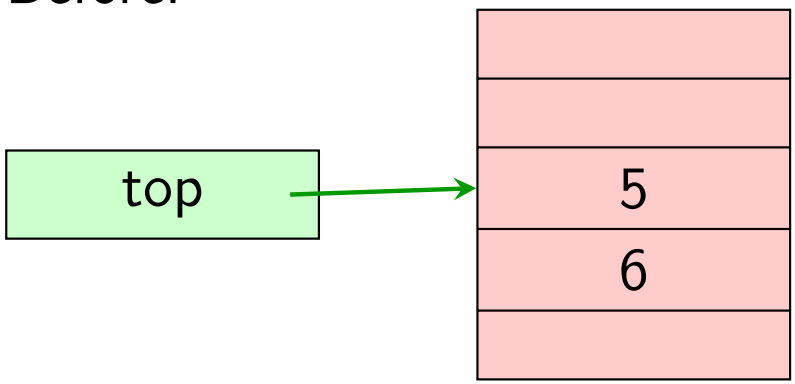
Naïve stack implementation

Suppose the VM has a stack in a hardware memory array, with a top-of-stack *pointer* in a hardware register. This is a zero-argument `stack_add` VM instruction:

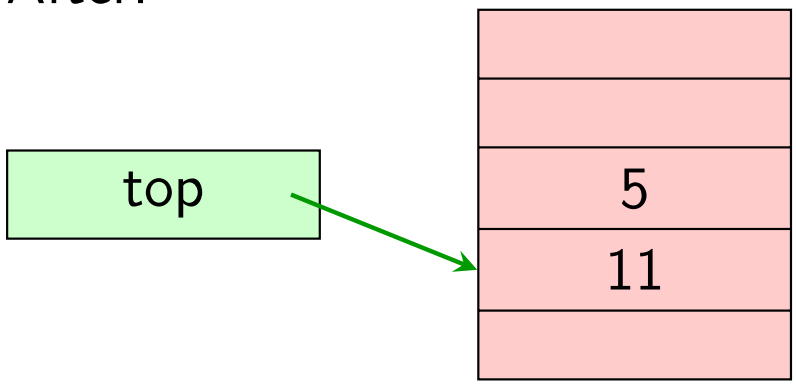
GNU C

```
label_stack_add:
  top [-1] = top [-1] + top [0];
  top --;
  /* Fallthru code omitted, same as always. */
```

Before:



After:



Two (independent) loads, one store. This looks better than our VM-register add: constant offsets from `top`, no index/offset loads.

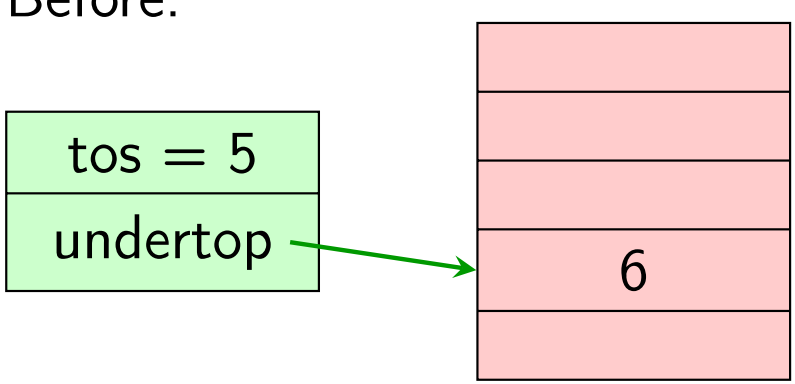
Top-Of-Stack (TOS) optimization

We can do even better: keep the VM top stack *element* in a hardware register (the rest of the stack still in a hardware memory array), and an under-top *pointer* in a second hardware register.

GNU C

```
label_stack_add:
  tos += undertop [0];
  undertop --;
  /* Fallthru code omitted, same as always. */
```

Before:



After:

Only one load. Other VM instructions working *only on the TOS* (for example `stack_increment`) require *zero* loads.

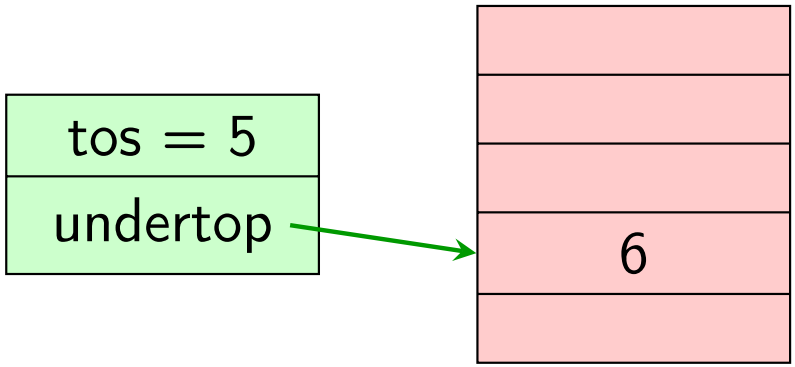
Top-Of-Stack (TOS) optimization

We can do even better: keep the VM top stack *element* in a hardware register (the rest of the stack still in a hardware memory array), and an under-top *pointer* in a second hardware register.

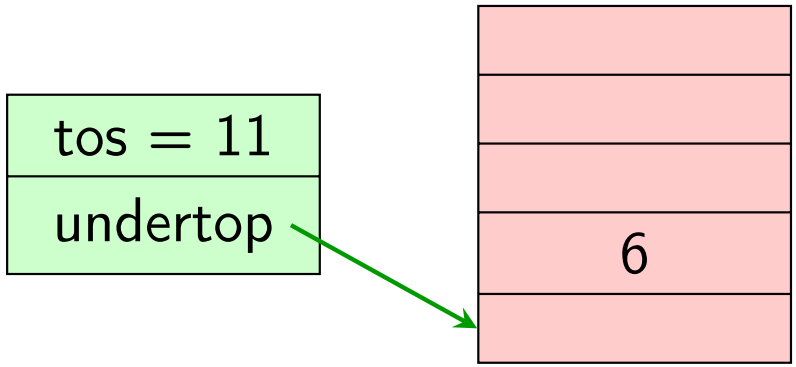
GNU C

```
label_stack_add:
  tos += undertop [0];
  undertop --;
  /* Fallthru code omitted, same as always. */
```

Before:



After:



Only one load. Other VM instructions working *only on the TOS* (for example `stack_increment`) require *zero* loads.

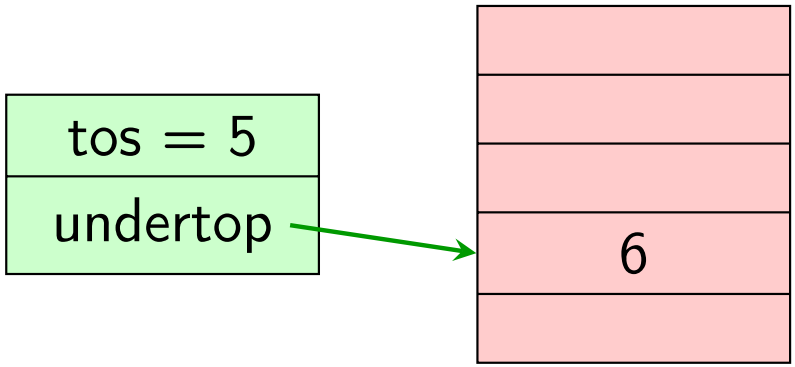
Top-Of-Stack (TOS) optimization

We can do even better: keep the VM top stack *element* in a hardware register (the rest of the stack still in a hardware memory array), and an under-top *pointer* in a second hardware register.

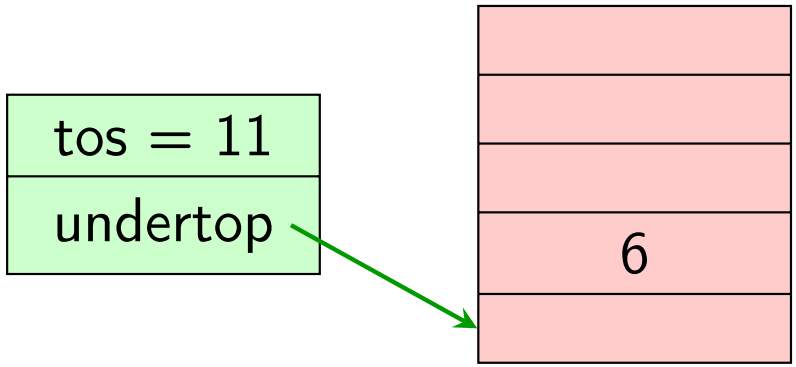
GNU C

```
label_stack_add:
  tos += undertop [0];
  undertop --;
  /* Fallthru code omitted, same as always. */
```

Before:



After:



Only one load. Other VM instructions working *only on the TOS* (for example `stack_increment`) require *zero* loads.

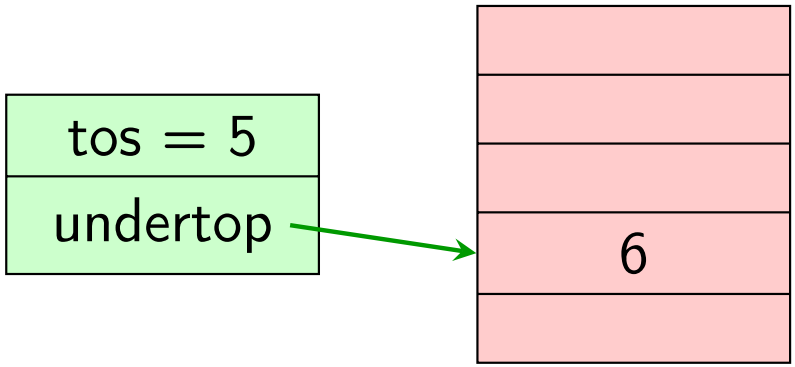
Top-Of-Stack (TOS) optimization

We can do even better: keep the VM top stack *element* in a hardware register (the rest of the stack still in a hardware memory array), and an under-top *pointer* in a second hardware register.

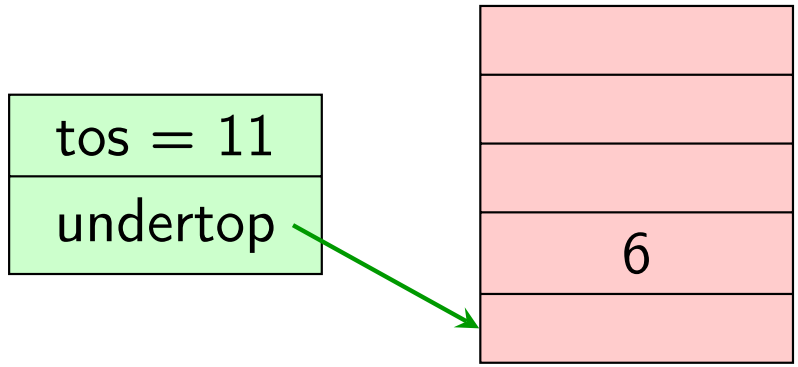
GNU C

```
label_stack_add:
  tos += undertop [0];
  undertop --;
  /* Fallthru code omitted, same as always. */
```

Before:



After:



Only one load. Other VM instructions working *only on the TOS* (for example `stack_increment`) require *zero* loads.

(Direct-threaded) TOS-optimized stack_add: operations

This includes the fallthru operations (*update insn*, *load target*, *jump*).

load target

↑

jump

update insn

update upder-top ptr.

load under-top

↑

add

Very "flat"-looking graph with short dependency chains (max length 1). Not many operations.

(Direct-threaded) TOS-optimized `stack_add`: operations

This includes the fallthru operations (*update insn*, *load target*, *jump*).

Very "flat"-looking graph with short dependency chains (max length 1). Not many operations.

(Direct-threaded) TOS-optimized stack_add: operations

This includes the fallthru operations (*update insn*, *load target*, *jump*).

Very "flat"-looking graph with short dependency chains (max length 1). Not many operations.

You've seen what every simple VMs does. . .

Nothing of what you saw up to here is new except for the removal of register *index shifts*, a minor optimization.

I want to make my VMs faster. In order of priority I need to:

- optimize VM register (and immediate argument) access [new]
- optimize fallthru [I learned the idea from [Ertl and Gregg, 2004], which builds upon previous work]
- remove `insn` and the VM program in memory [conceptually easy]
- optimize VM branches [my technique is new]

I want zero overhead in the common cases.

You've seen what every simple VMs does. . .

Nothing of what you saw up to here is new except for the removal of register *index shifts*, a minor optimization.

I want to make my VMs faster. In order of priority I need to:

- optimize VM register (and immediate argument) access [new]
- optimize fallthru [I learned the idea from [Ertl and Gregg, 2004], which builds upon previous work]
- remove `insn` and the VM program in memory [conceptually easy]
- optimize VM branches [my technique is new]

I want zero overhead in the common cases.

You've seen what every simple VMs does. . .

Nothing of what you saw up to here is new except for the removal of register *index shifts*, a minor optimization.

I want to make my VMs faster. In order of priority I need to:

- optimize VM register (and immediate argument) access [new]
- optimize fallthru [I learned the idea from [Ertl and Gregg, 2004], which builds upon previous work]
- remove `insn` and the VM program in memory [conceptually easy]
- optimize VM branches [my technique is new]

I want zero overhead in the common cases.

You've seen what every simple VMs does. . .

Nothing of what you saw up to here is new except for the removal of register *index shifts*, a minor optimization.

I want to make my VMs faster. In order of priority I need to:

- optimize VM register (and immediate argument) access [new]
- optimize fallthru [I learned the idea from [Ertl and Gregg, 2004], which builds upon previous work]
- remove `insn` and the VM program in memory [conceptually easy]
- optimize VM branches [my technique is new]

. . . I want zero overhead in the common cases.

You've seen what every simple VMs does. . .

Nothing of what you saw up to here is new except for the removal of register *index shifts*, a minor optimization.

I want to make my VMs faster. In order of priority I need to:

- optimize VM register (and immediate argument) access [new]
- optimize fallthru [I learned the idea from [Ertl and Gregg, 2004], which builds upon previous work]
- remove `insn` and the VM program in memory [conceptually easy]
- optimize VM branches [my technique is new]

. . . I want zero overhead in the common cases.

You've seen what every simple VMs does. . .

Nothing of what you saw up to here is new except for the removal of register *index shifts*, a minor optimization.

I want to make my VMs faster. In order of priority I need to:

- optimize VM register (and immediate argument) access [new]
- optimize fallthru [I learned the idea from [Ertl and Gregg, 2004], which builds upon previous work]
- remove `insn` and the VM program in memory [conceptually easy]
- optimize VM branches [my technique is new]

I want zero overhead in the common cases.

Optimizing VM register access

VM registers should not be in hardware memory.
I want them in hardware registers (as long as they fit).

The problem: every time I do anything with

`regs[e]`

and the value of e isn't known at compile time I lose. GCC can't put any `regs` element in a specific hardware register, while there is even one `regs[e]` expression with unknown e — reading or writing.

The solution: never use `regs[e]` with a non-constant e; or even split `regs` into scalar variables `reg_0`, `reg_1`, `reg_2`, ... and never take the address of those variables: writing "`& regs_i`" is forbidden for every i.

Optimizing VM register access

VM registers should not be in hardware memory.
I want them in hardware registers (as long as they fit).

The problem: every time I do anything with

$$\texttt{regs}[e]$$

and the value of e isn't known at compile time I lose. GCC can't put any `regs` element in a specific hardware register, while there is even one `regs[`e`]` expression with unknown e — reading or writing.

The solution: never use `regs[`e`]` with a non-constant e; or even split `regs` into scalar variables `reg_0`, `reg_1`, `reg_2`, ... and never take the address of those variables: writing "`&` `reg_`i" is forbidden for every i.

Optimizing VM register access

VM registers should not be in hardware memory.
I want them in hardware registers (as long as they fit).

The problem: every time I do anything with

`regs[e]`

and the value of *e* isn't known at compile time I lose. GCC can't put any `regs` element in a specific hardware register, while there is even one `regs[e]` expression with unknown *e* — reading or writing.

The solution: never use `regs[e]` with a non-constant *e*; or even split `regs` into scalar variables `reg_0`, `reg_1`, `reg_2`, ... and never take the address of those variables: writing "`& regs_i`" is forbidden for every *i*.

Let's look at a VM instruction such as add

[Here with register indices rather than offsets, just for simplicity: same point]

GNU C

```
label_add:
  regs[insn[3].i] = regs[insn[1].i] + regs[insn[2].i];
  insn += 4;
  goto * insn->label;
```

Here `regs` is (always) indexed with `insn[`k`].i`, an index coming from the interpreted program!

And this pattern is very common across VM instructions.

No hope with this VM instruction code.

Let's look at a VM instruction such as add

[Here with register indices rather than offsets, just for simplicity: same point]

GNU C

```
label_add:
  regs[insn[3].i] = regs[insn[1].i] + regs[insn[2].i];
  insn += 4;
  goto * insn->label;
```

Here `regs` is (always) indexed with `insn[`k`].i`, an index coming from the interpreted program!

And this pattern is very common across VM instructions.

No hope with this VM instruction code.

Let's look at a VM instruction such as add

[Here with register indices rather than offsets, just for simplicity: same point]

GNU C

```
label_add:
  regs[insn[3].i] = regs[insn[1].i] + regs[insn[2].i];
  insn += 4;
  goto * insn->label;
```

Here `regs` is (always) indexed with `insn[`k`].i`, an index coming from the interpreted program!

And this pattern is very common across VM instructions.

No hope with this VM instruction code.

Let's look at a VM instruction such as add

[Here with register indices rather than offsets, just for simplicity: same point]

GNU C

```
label_add:
  regs[insn[3].i] = regs[insn[1].i] + regs[insn[2].i];
  insn += 4;
  goto * insn->label;
```

Here `regs` is (always) indexed with `insn[`k`].i`, an index coming from the interpreted program!

And this pattern is very common across VM instructions.

No hope with this VM instruction code.

VM instruction specialization

A radical solution: forbid register indices/offsets as VM instruction arguments.

Remove the VM instruction `add` taking three index/offsets arguments from the interpreter. Instead there will be many *specialized* VM instructions:

`add/%r0/%r0/%r0`,
`add/%r0/%r0/%r1`,
`add/%r0/%r1/%r0`,
`add/%r0/%r1/%r1`, ...
`add/%r1/%r1/%r1`,
`add/%r0/%r0/%r2`, ... *Every possible combination.*

Specialized instructions have no register-index/offset arguments; the specializations of our example's `add` have all **zero** arguments.

VM instruction specialization

A radical solution: forbid register indices/offsets as VM instruction arguments.

Remove the VM instruction `add` taking three index/offsets arguments from the interpreter. Instead there will be many *specialized* VM instructions:

`add/%r0/%r0/%r0`,
`add/%r0/%r0/%r1`,
`add/%r0/%r1/%r0`,
`add/%r0/%r1/%r1`, ...
`add/%r1/%r1/%r1`,
`add/%r0/%r0/%r2`, ... *Every possible combination*.

Specialized instructions have no register-index/offset arguments; the specializations of our example's add have all **zero** arguments.

VM instruction specialization

A radical solution: forbid register indices/offsets as VM instruction arguments.

Remove the VM instruction `add` taking three index/offsets arguments from the interpreter. Instead there will be many *specialized* VM instructions:

`add/%r0/%r0/%r0`,
`add/%r0/%r0/%r1`,
`add/%r0/%r1/%r0`,
`add/%r0/%r1/%r1`, ...
`add/%r1/%r1/%r1`,
`add/%r0/%r0/%r2`, ... *Every possible combination*.

Specialized instructions have no register-index/offset arguments; the specializations of our example's `add` have all **zero** arguments.

Bear with me

Yes, I know that **you have objections** at this point.

Please give me one minute. I will address them.

Where am I going?

Specialization is not manageable in human-written code:

- very long and redundant code
- fragile with respect to trivial details [how many programs slot to skip for fallthru? The number depends on how many arguments are VM registers]

The solution is machine-generating C code.

Where am I going?

Specialization is not manageable in human-written code:

- very long and redundant code
- fragile with respect to trivial details [how many programs slot to skip for fallthru? The number depends on how many arguments are VM registers]

The solution is machine-generating C code.

Where am I going?

Specialization is not manageable in human-written code:

- very long and redundant code
- fragile with respect to trivial details [how many programs slot to skip for fallthru? The number depends on how many arguments are VM registers]

The solution is machine-generating C code.

The project takes shape

The new software I'm presenting is a code generator, automatically emitting C code for a VM from a human-written specification. Like Bison, and even more like Vmgen [Ertl et al., 2002], [Ertl, 2008].

- user-provided C code snippets for each unspecialized instruction
- convenient automatically-defined CPP macros to refer to (pre-specialization) arguments, and more
- fallthru code implicit for every VM instruction, automatically added by the generator

A VM instruction specification from the "Uninspired" VM (edited)

```
instruction add (?R, ?R, !R)
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGNO + UNINSPIRED_ARGN1;
  end
end
```

The project takes shape

The new software I'm presenting is a code generator, automatically emitting C code for a VM from a human-written specification. Like Bison, and even more like Vmgen [Ertl et al., 2002], [Ertl, 2008].

- user-provided C code snippets for each unspecialized instruction
- convenient automatically-defined CPP macros to refer to (pre-specialization) arguments, and more
- fallthru code implicit for every VM instruction, automatically added by the generator

A VM instruction specification from the "Uninspired" VM (edited)

```
instruction add (?R, ?R, !R)
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGNO + UNINSPIRED_ARGN1;
  end
end
```

The project takes shape

The new software I'm presenting is a code generator, automatically emitting C code for a VM from a human-written specification. Like Bison, and even more like Vmgen [Ertl et al., 2002], [Ertl, 2008].

- user-provided C code snippets for each unspecialized instruction
- convenient automatically-defined CPP macros to refer to (pre-specialization) arguments, and more
- fallthru code implicit for every VM instruction, automatically added by the generator

A VM instruction specification from the "Uninspired" VM (edited)

```
instruction add (?R, ?R, !R)
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGNO + UNINSPIRED_ARGN1;
  end
end
```

The project takes shape

The new software I'm presenting is a code generator, automatically emitting C code for a VM from a human-written specification. Like Bison, and even more like Vmgen [Ertl et al., 2002], [Ertl, 2008].

- user-provided C code snippets for each unspecialized instruction
- convenient automatically-defined CPP macros to refer to (pre-specialization) arguments, and more
- fallthru code implicit for every VM instruction, automatically added by the generator

A VM instruction specification from the "Uninspired" VM (edited)

```
instruction add (?R, ?R, !R)
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGNO + UNINSPIRED_ARGN1;
  end
end
```

The project takes shape

The new software I'm presenting is a code generator, automatically emitting C code for a VM from a human-written specification. Like Bison, and even more like Vmgen [Ertl et al., 2002], [Ertl, 2008].

- user-provided C code snippets for each unspecialized instruction
- convenient automatically-defined CPP macros to refer to (pre-specialization) arguments, and more
- fallthru code implicit for every VM instruction, automatically added by the generator

A VM instruction specification from the "Uninspired" VM (edited)

```
instruction add (?R, ?R, !R)
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGNO + UNINSPIRED_ARGN1;
  end
end
```

Generated C code: general

Making VMs general:

- VM registers, or stacks (TOS-optimized or not), both, anything else implemented by the user
- user-specified data types (register classes: for example *integer/pointer*, *floating point*, *vector*, . . .)
- several possible dispatching models
  - `switch`-dispatching, direct threading, other models I'll show later;
    - different performance profiles, identical behavior!
    - lots of `#ifdef`s in the generated C code; choose dispatching model by compiling with `-DDIRECT_THREADING`, . . .
- include custom C code from the user
- compatible with multi-threading and garbage collection, including *exact pointer-finding* [not just conservative as in Hans Bohem's GC]

Generated C code: general

Making VMs general:

- VM registers, or stacks (TOS-optimized or not), both, anything else implemented by the user
- user-specified data types (register classes: for example *integer/pointer*, *floating point*, *vector*, . . .)
- several possible dispatching models
  - `switch`-dispatching, direct threading, other models I'll show later;
    - different performance profiles, identical behavior!
    - lots of `#ifdef`s in the generated C code; choose dispatching model by compiling with `-DDIRECT_THREADING`, . . .
- include custom C code from the user
- compatible with multi-threading and garbage collection, including *exact pointer-finding* [not just conservative as in Hans Bohem's GC]

Generated C code: general

Making VMs general:

- VM registers, or stacks (TOS-optimized or not), both, anything else implemented by the user
- user-specified data types (register classes: for example *integer/pointer*, *floating point*, *vector*, . . .)
- several possible dispatching models
  - `switch`-dispatching, direct threading, other models I'll show later;
    - different performance profiles, identical behavior!
    - lots of `#ifdef`s in the generated C code; choose dispatching model by compiling with `-DDIRECT_THREADING`, . . .
- include custom C code from the user
- compatible with multi-threading and garbage collection, including *exact pointer-finding* [not just conservative as in Hans Bohem's GC]

Generated C code: general

Making VMs general:

- VM registers, or stacks (TOS-optimized or not), both, anything else implemented by the user
- user-specified data types (register classes: for example *integer/pointer*, *floating point*, *vector*, . . .)
- several possible dispatching models
  - `switch`-dispatching, direct threading, other models I'll show later;
    - different performance profiles, identical behavior!
    - lots of `#ifdef`s in the generated C code; choose dispatching model by compiling with `-DDIRECT_THREADING`, . . .
- include custom C code from the user
- compatible with multi-threading and garbage collection, including *exact pointer-finding* [not just conservative as in Hans Bohem's GC]

Generated C code: general

Making VMs general:

- VM registers, or stacks (TOS-optimized or not), both, anything else implemented by the user
- user-specified data types (register classes: for example *integer/pointer*, *floating point*, *vector*, . . .)
- several possible dispatching models
  - `switch`-dispatching, direct threading, other models I'll show later;
    - different performance profiles, identical behavior!
    - lots of `#ifdef`s in the generated C code; choose dispatching model by compiling with `-DDIRECT_THREADING`, . . .
- include custom C code from the user
- compatible with multi-threading and garbage collection, including *exact pointer-finding* [not just conservative as in Hans Bohem's GC]

Generated C code: portable

Making VMs portable with respect to different CPU architectures (also important for political reasons: free hardware as a prerequisite for privacy)

- Using C with as little assembly as possible, and not in user code (the assembly part is VM-independent, and already provided)
- even that little assembly is optional, only for better performance
  - VMs behave identically, with or without assembly snippets
  - direct threading with specialization is as portable as C (it's switch-dispatching even more portable (no goto), but was not implemented, but trivial)
- compiled VMs work comfortably even on "small" machines (32MB RAM is plenty; probably 8 or even 4MB is enough)
  - (*Compiling VMs* is heavier, as you have guessed already)

Generated C code: portable

Making VMs portable with respect to different CPU architectures (also important for political reasons: free hardware as a prerequisite for privacy)

- Using C with as little assembly as possible, and not in user code (the assembly part is VM-independent, and already provided)
- even that little assembly is optional, only for better performance
  - VMs behave identically, with or without assembly support
  - direct threading with specialization is as portable as GCC
  - switch-dispatching even more portable (no goto *) (not yet implemented, but trivial)
- compiled VMs work comfortably even on "small" machines (32MB RAM is plenty; probably 8 or even 4MB is enough)
  - (*Compiling VMs* is heavier, as you have guessed already)

Generated C code: portable

Making VMs portable with respect to different CPU architectures (also important for political reasons: free hardware as a prerequisite for privacy)

- Using C with as little assembly as possible, and not in user code (the assembly part is VM-independent, and already provided)
- even that little assembly is optional, only for better performance
  - VMs behave identically, with or without assembly support
  - direct threading with specialization is as portable as GCC
  - `switch`-dispatching even more portable (no `goto *`) (not yet implemented, but trivial)
- compiled VMs work comfortably even on "small" machines (32MB RAM is plenty; probably 8 or even 4MB is enough)
  - (*Compiling VMs* is heavier, as you have guessed already)

Generated C code: portable

Making VMs portable with respect to different CPU architectures (also important for political reasons: free hardware as a prerequisite for privacy)

- Using C with as little assembly as possible, and not in user code (the assembly part is VM-independent, and already provided)
- even that little assembly is optional, only for better performance
  - VMs behave identically, with or without assembly support
  - direct threading with specialization is as portable as GCC
  - `switch`-dispatching even more portable (no `goto *`) (not yet implemented, but trivial)
- compiled VMs work comfortably even on "small" machines (32MB RAM is plenty; probably 8 or even 4MB is enough)
  - (*Compiling VMs* is heavier, as you have guessed already)

Generated C code: portable

Making VMs portable with respect to different CPU architectures (also important for political reasons: free hardware as a prerequisite for privacy)

- Using C with as little assembly as possible, and not in user code (the assembly part is VM-independent, and already provided)
- even that little assembly is optional, only for better performance
  - VMs behave identically, with or without assembly support
  - direct threading with specialization is as portable as GCC
  - `switch`-dispatching even more portable (no `goto *`) (not yet implemented, but trivial)
- compiled VMs work comfortably even on "small" machines (32MB RAM is plenty; probably 8 or even 4MB is enough)
  - (*Compiling VMs* is heavier, as you have guessed already)

Generated C code: portable

Making VMs portable with respect to different CPU architectures (also important for political reasons: free hardware as a prerequisite for privacy)

- Using C with as little assembly as possible, and not in user code (the assembly part is VM-independent, and already provided)
- even that little assembly is optional, only for better performance
  - VMs behave identically, with or without assembly support
  - direct threading with specialization is as portable as GCC
  - `switch`-dispatching even more portable (no `goto *`) (not yet implemented, but trivial)
- compiled VMs work comfortably even on "small" machines (32MB RAM is plenty; probably 8 or even 4MB is enough)
  - (*Compiling VMs* is heavier, as you have guessed already)

Generated C code: portable

Making VMs portable with respect to different CPU architectures (also important for political reasons: free hardware as a prerequisite for privacy)

- Using C with as little assembly as possible, and not in user code (the assembly part is VM-independent, and already provided)
- even that little assembly is optional, only for better performance
  - VMs behave identically, with or without assembly support
  - direct threading with specialization is as portable as GCC
  - `switch`-dispatching even more portable (no `goto *`) (not yet implemented, but trivial)
- compiled VMs work comfortably even on "small" machines (32MB RAM is plenty; probably 8 or even 4MB is enough)
  - (*Compiling VMs* is heavier, as you have guessed already)

Generated C code: portable

Making VMs portable with respect to different CPU architectures (also important for political reasons: free hardware as a prerequisite for privacy)

- Using C with as little assembly as possible, and not in user code (the assembly part is VM-independent, and already provided)
- even that little assembly is optional, only for better performance
  - VMs behave identically, with or without assembly support
  - direct threading with specialization is as portable as GCC
  - `switch`-dispatching even more portable (no `goto *`) (not yet implemented, but trivial)
- compiled VMs work comfortably even on "small" machines (32MB RAM is plenty; probably 8 or even 4MB is enough)
  - (*Compiling VMs* is heavier, as you have guessed already)

Generated-code goodies

Along with the generated code you get:

- C API for dynamically generating and executing VM programs from your application
- driver with command-line options (`main` with convenient GNU command-line support for debugging and benchmarking)
- frontend: VM program parser and printer
- cross-compilation support
- disassembly to native or (via `qemu-user`) cross- code
- testsuite (even cross-, via `qemu-user`)

If you want to generate a direct-threaded VM of the kind many projects already use (Emacs, Guile, ...), it's trivial. But getting a much more efficient VM is not any more difficult.

Generated-code goodies

Along with the generated code you get:

- C API for dynamically generating and executing VM programs from your application
- driver with command-line options (`main` with convenient GNU command-line support for debugging and benchmarking)
- frontend: VM program parser and printer
- cross-compilation support
- disassembly to native or (via `qemu-user`) cross- code
- testsuite (even cross-, via `qemu-user`)

If you want to generate a direct-threaded VM of the kind many projects already use (Emacs, Guile, ...), it's trivial. But getting a much more efficient VM is not any more difficult.

Generated-code goodies

Along with the generated code you get:

- C API for dynamically generating and executing VM programs from your application
- driver with command-line options (`main` with convenient GNU command-line support for debugging and benchmarking)
- frontend: VM program parser and printer
- cross-compilation support
- disassembly to native or (via `qemu-user`) cross- code
- testsuite (even cross-, via `qemu-user`)

If you want to generate a direct-threaded VM of the kind many projects already use (Emacs, Guile, ...), it's trivial. But getting a much more efficient VM is not any more difficult.

Generated-code goodies

Along with the generated code you get:

- C API for dynamically generating and executing VM programs from your application
- driver with command-line options (`main` with convenient GNU command-line support for debugging and benchmarking)
- frontend: VM program parser and printer
- cross-compilation support
- disassembly to native or (via `qemu-user`) cross- code
- testsuite (even cross-, via `qemu-user`)

If you want to generate a direct-threaded VM of the kind many projects already use (Emacs, Guile, ...), it's trivial. But getting a much more efficient VM is not any more difficult.

Generated-code goodies

Along with the generated code you get:

- C API for dynamically generating and executing VM programs from your application
- driver with command-line options (`main` with convenient GNU command-line support for debugging and benchmarking)
- frontend: VM program parser and printer
- cross-compilation support
- disassembly to native or (via `qemu-user`) cross- code
- testsuite (even cross-, via `qemu-user`)

If you want to generate a direct-threaded VM of the kind many projects already use (Emacs, Guile, ...), it's trivial. But getting a much more efficient VM is not any more difficult.

Generated-code goodies

Along with the generated code you get:

- C API for dynamically generating and executing VM programs from your application
- driver with command-line options (`main` with convenient GNU command-line support for debugging and benchmarking)
- frontend: VM program parser and printer
- cross-compilation support
- disassembly to native or (via `qemu-user`) cross- code
- testsuite (even cross-, via `qemu-user`)

If you want to generate a direct-threaded VM of the kind many projects already use (Emacs, Guile, ...), it's trivial. But getting a much more efficient VM is not any more difficult.

Generated-code goodies

Along with the generated code you get:

- C API for dynamically generating and executing VM programs from your application
- driver with command-line options (`main` with convenient GNU command-line support for debugging and benchmarking)
- frontend: VM program parser and printer
- cross-compilation support
- disassembly to native or (via `qemu-user`) cross- code
- testsuite (even cross-, via `qemu-user`)

... If you want to generate a direct-threaded VM of the kind many projects already use (Emacs, Guile, ...), it's trivial. But getting a much more efficient VM is not any more difficult.

Generated-code goodies

Along with the generated code you get:

- C API for dynamically generating and executing VM programs from your application
- driver with command-line options (`main` with convenient GNU command-line support for debugging and benchmarking)
- frontend: VM program parser and printer
- cross-compilation support
- disassembly to native or (via `qemu-user`) cross- code
- testsuite (even cross-, via `qemu-user`)

If you want to generate a direct-threaded VM of the kind many projects already use (Emacs, Guile, . . .), it's trivial. But getting a much more efficient VM is not any more difficult.

VM specialized instructions: combinatorial explosion?

If we have n registers and m instructions (for example) all taking 3 register indices as arguments, specialized instructions are $m \cdot n^3$.

Yes, there are practical limits on how many VM registers of this kind you can have.

There are ways to reduce this growth and some optimizations I haven't implemented yet, but compiling a machine-generated VM is heavy. GCC can use GBs of RAM and take minutes to run when VM registers are many.

VM specialized instructions: combinatorial explosion?

If we have n registers and m instructions (for example) all taking 3 register indices as arguments, specialized instructions are $m \cdot n^3$.

Yes, there are practical limits on how many VM registers of this kind you can have.

There are ways to reduce this growth and some optimizations I haven't implemented yet, but compiling a machine-generated VM is heavy. GCC can use GBs of RAM and take minutes to run when VM registers are many.

Limiting combinatorial explosion

Some specialized instructions are useless or can be normalized:

- For example, addition is commutative: `add/%r0/%r1/%r2` and `add/%r1/%r0/%r2` do the same work, and we can keep only one. This halves the number of (commutative) specialized instructions.
- We can also rewrite every specialized instruction such as

  `add/%ri/%rj/%rk`

  into a two-specialized-instruction sequence

  `copy/%rj/%rk`
  `add/%ri/%rk/%rk`

  whenever $j \neq k$. [This is correct because `add` writes its third argument, but doesn't read it.] This rewrite can cut the number of specialized instructions from $m \cdot n^3$ to $m \cdot n^2$.

Every specialized instruction which is not a rewrite target "doesn't exist".

Limiting combinatorial explosion

Some specialized instructions are useless or can be normalized:

- For example, addition is commutative: `add/%r0/%r1/%r2` and `add/%r1/%r0/%r2` do the same work, and we can keep only one. This halves the number of (commutative) specialized instructions.
- We can also rewrite every specialized instruction such as

  `add/%ri/%rj/%rk`

  into a two-specialized-instruction sequence

  `copy/%rj/%rk`
  `add/%ri/%rk/%rk`

  whenever $j \neq k$. [This is correct because add writes its third argument, but doesn't read it.] This rewrite can cut the number of specialized instructions from $m \cdot n^3$ to $m \cdot n^2$.

Every specialized instruction which is not a rewrite target "doesn't exist".

Limiting combinatorial explosion

Some specialized instructions are useless or can be normalized:

- For example, addition is commutative: `add/%r0/%r1/%r2` and `add/%r1/%r0/%r2` do the same work, and we can keep only one. This halves the number of (commutative) specialized instructions.
- We can also rewrite every specialized instruction such as

  $$\texttt{add/\%r}i\texttt{/\%r}j\texttt{/\%r}k$$

  into a two-specialized-instruction sequence

  $$\texttt{copy/\%r}j\texttt{/\%r}k$$
  $$\texttt{add/\%r}i\texttt{/\%r}k\texttt{/\%r}k$$

  whenever $j \neq k$. [This is correct because `add` writes its third argument, but doesn't read it.] This rewrite can cut the number of specialized instructions from $m \cdot n^3$ to $m \cdot n^2$.

Every specialized instruction which is not a rewrite target "doesn't exist".

Limiting combinatorial explosion

Some specialized instructions are useless or can be normalized:

- For example, addition is commutative: `add/%r0/%r1/%r2` and `add/%r1/%r0/%r2` do the same work, and we can keep only one. This halves the number of (commutative) specialized instructions.
- We can also rewrite every specialized instruction such as

  $$\texttt{add/\%r}i\texttt{/\%r}j\texttt{/\%r}k$$

  into a two-specialized-instruction sequence

  $$\texttt{copy/\%r}j\texttt{/\%r}k$$
  $$\texttt{add/\%r}i\texttt{/\%r}k\texttt{/\%r}k$$

  whenever $j \neq k$. [This is correct because `add` writes its third argument, but doesn't read it.] This rewrite can cut the number of specialized instructions from $m \cdot n^3$ to $m \cdot n^2$.

Every specialized instruction which is not a rewrite target "doesn't exist".

Limiting combinatorial explosion: rewriting

What I've outlined can be expressed as a rewriting system.

Which rewrites are valid depends on the properties of each specific instruction: such properties must be declared by the user in her VM specification, and cannot in general be inferred.

I've not fully implemented rewriting yet, even if the parser recognizes a preliminary syntax. I want a rule-based system which is expressive enough to limit growth, and also to perform a few optimizations in the VM program [for this reason I will implement rewriting on *unspecialized* VM instructions]

Some manual tests have convinced me that with fewer useless VM instructions GCC will do a better job of allocating registers for those which remain. Implementing rewriting is high-priority.

[GCC register allocation gets worse with many VM registers, on most *but not all* architectures. Is there a GCC expert I can talk to here?]

Limiting combinatorial explosion: rewriting

What I've outlined can be expressed as a rewriting system.

Which rewrites are valid depends on the properties of each specific instruction: such properties must be declared by the user in her VM specification, and cannot in general be inferred.

I've not fully implemented rewriting yet, even if the parser recognizes a preliminary syntax. I want a rule-based system which is expressive enough to limit growth, and also to perform a few optimizations in the VM program [for this reason I will implement rewriting on *unspecialized* VM instructions]

Some manual tests have convinced me that with fewer useless VM instructions GCC will do a better job of allocating registers for those which remain. Implementing rewriting is high-priority.

[GCC register allocation gets worse with many VM registers, on most *but not all* architectures. Is there a GCC expert I can talk to here?]

Limiting combinatorial explosion: rewriting

What I've outlined can be expressed as a rewriting system.

Which rewrites are valid depends on the properties of each specific instruction: such properties must be declared by the user in her VM specification, and cannot in general be inferred.

I've not fully implemented rewriting yet, even if the parser recognizes a preliminary syntax. I want a rule-based system which is expressive enough to limit growth, and also to perform a few optimizations in the VM program [for this reason I will implement rewriting on *unspecialized* VM instructions]

Some manual tests have convinced me that with fewer useless VM instructions GCC will do a better job of allocating registers for those which remain. Implementing rewriting is high-priority.

[GCC register allocation gets worse with many VM registers, on most *but not all* architectures. Is there a GCC expert I can talk to here?]

Limiting combinatorial explosion: rewriting

What I've outlined can be expressed as a rewriting system.

Which rewrites are valid depends on the properties of each specific instruction: such properties must be declared by the user in her VM specification, and cannot in general be inferred.

I've not fully implemented rewriting yet, even if the parser recognizes a preliminary syntax. I want a rule-based system which is expressive enough to limit growth, and also to perform a few optimizations in the VM program [for this reason I will implement rewriting on *unspecialized* VM instructions]

Some manual tests have convinced me that with fewer useless VM instructions GCC will do a better job of allocating registers for those which remain. Implementing rewriting is high-priority.

[GCC register allocation gets worse with many VM registers, on most *but not all* architectures. Is there a GCC expert I can talk to here?]

Combinatorial explosion and stack-based instructions

Do we have the same combinatorial explosion problem with stack-based instruction?

- No. The unspecialized VM instruction `add_stack` has *zero arguments*, and only *one specialization*.
  - More in general implied operands limit combinatorial explosion, even with registers. *Example: special-purpose registers:* `mul` *and* `div` *could always write to the same destination register . . .*
- Rewrite rules are an easy and powerful way of optimizing stack code.

  Example:

```
stack_push 10
stack_plus
```

→

```
stack_plusi 10
```

We'll see how effective this is after I implement rewriting.

Combinatorial explosion and stack-based instructions

Do we have the same combinatorial explosion problem with stack-based instruction?

- No. The unspecialized VM instruction `add_stack` has *zero arguments*, and only *one specialization*.
  - More in general implied operands limit combinatorial explosion, even with registers. *Example: special-purpose registers:* `mul` *and* `div` *could always write to the same destination register . . .*
- Rewrite rules are an easy and powerful way of optimizing stack code.

  Example:

```
stack_push 10
stack_plus
```

→

```
stack_plusi 10
```

We'll see how effective this is after I implement rewriting.

Combinatorial explosion and stack-based instructions

Do we have the same combinatorial explosion problem with stack-based instruction?

- No. The unspecialized VM instruction `add_stack` has *zero arguments*, and only *one specialization*.
  - More in general implied operands limit combinatorial explosion, even with registers. *Example: special-purpose registers:* `mul` *and* `div` *could always write to the same destination register . . .*
- Rewrite rules are an easy and powerful way of optimizing stack code.

  Example:

```
stack_push 10
stack_plus
```

  →

```
stack_plusi 10
```

We'll see how effective this is after I implement rewriting.

Combinatorial explosion and stack-based instructions

Do we have the same combinatorial explosion problem with stack-based instruction?

- No. The unspecialized VM instruction `add_stack` has *zero arguments*, and only *one specialization*.
  - More in general implied operands limit combinatorial explosion, even with registers. *Example: special-purpose registers:* `mul` *and* `div` *could always write to the same destination register . . .*
- Rewrite rules are an easy and powerful way of optimizing stack code.
  Example:

```
stack_push 10
stack_plus
```

→

```
stack_plusi 10
```

We'll see how effective this is after I implement rewriting.

Is VM specialization worth the trouble?

Remove every access to `regs` with a non-constant index from the interpreter. Then:

(Macro-expanded) GNU C

```
label_add_r0_r1_r1:
  regs[1] = regs[0] + regs[1];
  insn ++; // skip code ptr. only
  goto * insn->label;
```

compiled (x86_64)

```
addq $8, %rax
addq %rbx, %rcx
jmpq *(%rax)  # Jump via memory
```

Much better than the unspecialized version!

Now `regs` indices are constants (different in every specialization):

Here GCC has kept the VM register `%r0` in the hardware register `%rbx` and the VM register `%r1` in the hardware register `%rcx`.

[When there aren't enough hardware machine registers GCC will allocate some VM registers on the C stack, at a known offset from the C stack/frame pointer: still faster than without specialization.]

Is VM specialization worth the trouble?

Remove every access to `regs` with a non-constant index from the interpreter. Then:

(Macro-expanded) GNU C

```
label_add_r0_r1_r1:
  regs[1] = regs[0] + regs[1];
  insn ++; // skip code ptr. only
  goto * insn->label;
```

compiled (x86_64)

```
addq $8, %rax
addq %rbx, %rcx
jmpq *(%rax)   # Jump via memory
```

Now `regs` indices are constants (different in every specialization):

Much better than the unspecialized version!

Here GCC has kept the VM register `%r0` in the hardware register `%rbx` and the VM register `%r1` in the hardware register `%rcx`.

[When there aren't enough hardware machine registers GCC will allocate some VM registers on the C stack, at a known offset from the C stack/frame pointer: still faster than without specialization.]

Is VM specialization worth the trouble?

Remove every access to `regs` with a non-constant index from the interpreter. Then:

(Macro-expanded) GNU C

```
label_add_r0_r1_r1:
  regs[1] = regs[0] + regs[1];
  insn ++; // skip code ptr. only
  goto * insn->label;
```

compiled (x86_64)

```
addq $8, %rax
addq %rbx, %rcx
jmpq *(%rax)   # Jump via memory
```

Now `regs` indices are constants (different in every specialization):

Much better than the unspecialized version!

Here GCC has kept the VM register `%r0` in the hardware register `%rbx` and the VM register `%r1` in the hardware register `%rcx`.

[When there aren't enough hardware machine registers GCC will allocate some VM registers on the C stack, at a known offset from the C stack/frame pointer: still faster than without specialization.]

Is VM specialization worth the trouble?

Remove every access to `regs` with a non-constant index from the interpreter. Then:

(Macro-expanded) GNU C

```
label_add_r0_r1_r1:
  regs[1] = regs[0] + regs[1];
  insn ++; // skip code ptr. only
  goto * insn->label;
```

compiled (x86_64)

```
addq $8, %rax
addq %rbx, %rcx
jmpq *(%rax)   # Jump via memory
```

Now `regs` indices are constants (different in every specialization):

Much better than the unspecialized version!

Here GCC has kept the VM register `%r0` in the hardware register `%rbx` and the VM register `%r1` in the hardware register `%rcx`.

[When there aren't enough hardware machine registers GCC will allocate some VM registers on the C stack, at a known offset from the C stack/frame pointer: still faster than without specialization.]

More on specialization: slow VM registers

There's a limit to the number of VM registers we can use for generating specialized instruction. However, for convenience and expressiveness, we can *also*, optionally, provide an unlimited number of additional VM registers, less efficient to access.

We call the VM registers on which we specialize fast registers, and the others slow registers. Slow registers are implemented as a (separate) array in hardware memory, exactly like pre-specialization VM registers, pointed by `slow_regs`.

The distinction between fast and slow registers is transparent:

A VM instruction specification from the "Uninspired" VM (edited)

```
instruction add (?R, ?R, !R)  # Each 'R' can be fast or slow
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGN0 + UNINSPIRED_ARGN1;
  end
end
```

More on specialization: slow VM registers

There's a limit to the number of VM registers we can use for generating specialized instruction. However, for convenience and expressiveness, we can *also*, optionally, provide an unlimited number of additional VM registers, less efficient to access.

We call the VM registers on which we specialize fast registers, and the others slow registers. Slow registers are implemented as a (separate) array in hardware memory, exactly like pre-specialization VM registers, pointed by `slow_regs`.

The distinction between fast and slow registers is transparent:

A VM instruction specification from the "Uninspired" VM (edited)

```
instruction add (?R, ?R, !R)  # Each 'R' can be fast or slow
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGNO + UNINSPIRED_ARGN1;
  end
end
```

More on specialization: slow VM registers

There's a limit to the number of VM registers we can use for generating specialized instruction. However, for convenience and expressiveness, we can *also*, optionally, provide an unlimited number of additional VM registers, less efficient to access.

We call the VM registers on which we specialize fast registers, and the others slow registers. Slow registers are implemented as a (separate) array in hardware memory, exactly like pre-specialization VM registers, pointed by `slow_regs`.

The distinction between fast and slow registers is transparent:

A VM instruction specification from the "Uninspired" VM (edited)

```
instruction add (?R, ?R, !R)  # Each 'R' can be fast or slow
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGN0 + UNINSPIRED_ARGN1;
  end
end
```

Slow VM registers: generated code expansion

The same VM instruction can indifferently use fast or slow VM registers, or mix them together, according to each specialization:

(Macro-expanded) GNU C

```
label_add_r0_rR_r0:
  regs[0] = regs[0] + (* (long *) (slow_regs + insn[1].i));
  insn += 2;  // skip code ptr. and the residual slow_regs offt.
  goto * insn->label;
```

The generator always encodes slow VM register arguments as pre-shifted offsets from `slow_regs` within the VM program (here `insn[1].i`).

Reading a VM slow register value still takes two inter-dependent loads.

Slow VM registers: generated code expansion

The same VM instruction can indifferently use fast or slow VM registers, or mix them together, according to each specialization:

(Macro-expanded) GNU C

```
label_add_r0_rR_r0:
  regs[0] = regs[0] + (* (long *) (slow_regs + insn[1].i));
  insn += 2;  // skip code ptr. and the residual slow_regs offt.
  goto * insn->label;
```

The generator always encodes slow VM register arguments as pre-shifted offsets from `slow_regs` within the VM program (here `insn[1].i`).

Reading a VM slow register value still takes two inter-dependent loads.

Slow VM registers: generated code expansion

The same VM instruction can indifferently use fast or slow VM registers, or mix them together, according to each specialization:

(Macro-expanded) GNU C

```
label_add_r0_rR_r0:
  regs[0] = regs[0] + (* (long *) (slow_regs + insn[1].i));
  insn += 2;  // skip code ptr. and the residual slow_regs offt.
  goto * insn->label;
```

The generator always encodes slow VM register arguments as pre-shifted offsets from `slow_regs` within the VM program (here `insn[1].i`).

Reading a VM slow register value still takes two inter-dependent loads.

More on specialization: literals

We can specialize on a set of particular instruction literal arguments as well.

The same instruction can also be made to access either a register or a literal at some position. For example adding 1 and -1 to a VM register is presumably common:

VM instruction specification from the "Uninspired" VM

```
instruction add (?Rn 1 -1, ?Rn 1 -1, !R)
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGN0 + UNINSPIRED_ARGN1;
  end
end
```

Specialized literals are not held in the VM program (not "*residualized*").

More on specialization: literals

We can specialize on a set of particular instruction literal arguments as well.

The same instruction can also be made to access either a register or a literal at some position. For example adding 1 and -1 to a VM register is presumably common:

VM instruction specification from the "Uninspired" VM

```
instruction add (?Rn 1 -1, ?Rn 1 -1, !R)
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGN0 + UNINSPIRED_ARGN1;
  end
end
```

Specialized literals are not held in the VM program (not "*residualized*").

More on specialization: literals

We can specialize on a set of particular instruction literal arguments as well.

The same instruction can also be made to access either a register or a literal at some position. For example adding 1 and -1 to a VM register is presumably common:

VM instruction specification from the "Uninspired" VM

```
instruction add (?Rn 1 -1, ?Rn 1 -1, !R)
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGN0 + UNINSPIRED_ARGN1;
  end
end
```

Specialized literals are not held in the VM program (not "*residualized*").

More on specialization: literals

We can specialize on a set of particular instruction literal arguments as well.

The same instruction can also be made to access either a register or a literal at some position. For example adding 1 and -1 to a VM register is presumably common:

VM instruction specification from the "Uninspired" VM

```
instruction add (?Rn 1 -1, ?Rn 1 -1, !R)
  code
    UNINSPIRED_ARGN2 = UNINSPIRED_ARGN0 + UNINSPIRED_ARGN1;
  end
end
```

Specialized literals are not held in the VM program (not "*residualized*").

More on specialization: literals performance

Now `regs` indices are constant, and literal constants are substituted into the VM instruction code in C.

GNU C (Macro-expanded)

```
label_addi_r0_n1_r0:
  regs[0] = regs[0] + 1;
  insn ++; // skip code ptr. only
  goto * insn->label;
```

compiled (x86_64)

```
addq $8, %rax
addq $1, %rbx
jmpq *(%rax)  # Jump via memory
```

Good!

Here GCC emitted `$1` as a hardware instruction immediate. This code reads L1d only in the fallthru part.

[Always possible with small constants on most architectures]

More on specialization: literals performance

Now `regs` indices are constant, and literal constants are substituted into the VM instruction code in C.

GNU C (Macro-expanded)

```
label_addi_r0_n1_r0:
  regs[0] = regs[0] + 1;
  insn ++; // skip code ptr. only
  goto * insn->label;
```

compiled (x86_64)

```
addq $8, %rax
addq $1, %rbx
jmpq *(%rax)   # Jump via memory
```

Good!

Here GCC emitted `$1` as a hardware instruction immediate. This code reads L1d only in the fallthru part.

[Always possible with small constants on most architectures]

More on specialization: literals performance

Now `regs` indices are constant, and literal constants are substituted into the VM instruction code in C.

GNU C (Macro-expanded)

```
label_addi_r0_n1_r0:
  regs[0] = regs[0] + 1;
  insn ++; // skip code ptr. only
  goto * insn->label;
```

compiled (x86_64)

```
addq $8, %rax
addq $1, %rbx
jmpq *(%rax)   # Jump via memory
```

Good!

Here GCC emitted `$1` as a hardware instruction immediate. This code reads L1d only in the fallthru part.

[Always possible with small constants on most architectures]

More on specialization: literals performance

Now `regs` indices are constant, and literal constants are substituted into the VM instruction code in C.

GNU C (Macro-expanded)

```
label_addi_r0_n1_r0:
  regs[0] = regs[0] + 1;
  insn ++; // skip code ptr. only
  goto * insn->label;
```

compiled (x86_64)

```
addq $8, %rax
addq $1, %rbx
jmpq *(%rax)   # Jump via memory
```

Good!

Here GCC emitted `$1` as a hardware instruction immediate. This code reads L1d only in the fallthru part.

[Always possible with small constants on most architectures]

VM operand access is now fast in the common case

solved!

[Little demo of the Uninspired VM, with direct-threaded dispatching and specialization for fast operand access]

VM operand access is now fast in the common case

solved!

[Little demo of the Uninspired VM, with direct-threaded dispatching and specialization for fast operand access]

The next bottleneck

We have solved the problem of operand access in the common case.

The interpreter bottleneck has moved: now the problem is dispatching.

- the fallthru code at the end of the typical VM instruction now takes longer than the part doing useful work.
- VM branches are less common than falling thru in real-world programs (the down-counter example is not representative)
  - . . . so let's not think about VM branches yet

The next bottleneck

We have solved the problem of operand access in the common case.

The interpreter bottleneck has moved: now the problem is dispatching.

- the fallthru code at the end of the typical VM instruction now takes longer than the part doing useful work.
- VM branches are less common than falling thru in real-world programs (the down-counter example is not representative)
  - . . . so let's not think about VM branches yet

VM instruction replication

All VM instructions but unconditional branches end with slow fallthru code. We want to remove it.

The solution is copying compiled specialized VM instruction code sequences one after another, concatenating them into hardware machine-code basic blocks. Then each VM instruction in the block automatically "falls thru" into the next.

A code pointer is only needed at the beginning of each basic block.

I call this dispatching style minimal threading: it's an optimization of direct threading.

VM instruction replication

All VM instructions but unconditional branches end with slow fallthru code. We want to remove it.

The solution is copying compiled specialized VM instruction code sequences one after another, concatenating them into hardware machine-code basic blocks. Then each VM instruction in the block automatically "falls thru" into the next.

A code pointer is only needed at the beginning of each basic block.

I call this dispatching style minimal threading: it's an optimization of direct threading.

VM instruction replication

All VM instructions but unconditional branches end with slow fallthru code. We want to remove it.

The solution is copying compiled specialized VM instruction code sequences one after another, concatenating them into hardware machine-code basic blocks. Then each VM instruction in the block automatically "falls thru" into the next.

A code pointer is only needed at the beginning of each basic block.

I call this dispatching style minimal threading: it's an optimization of direct threading.

VM instruction replication

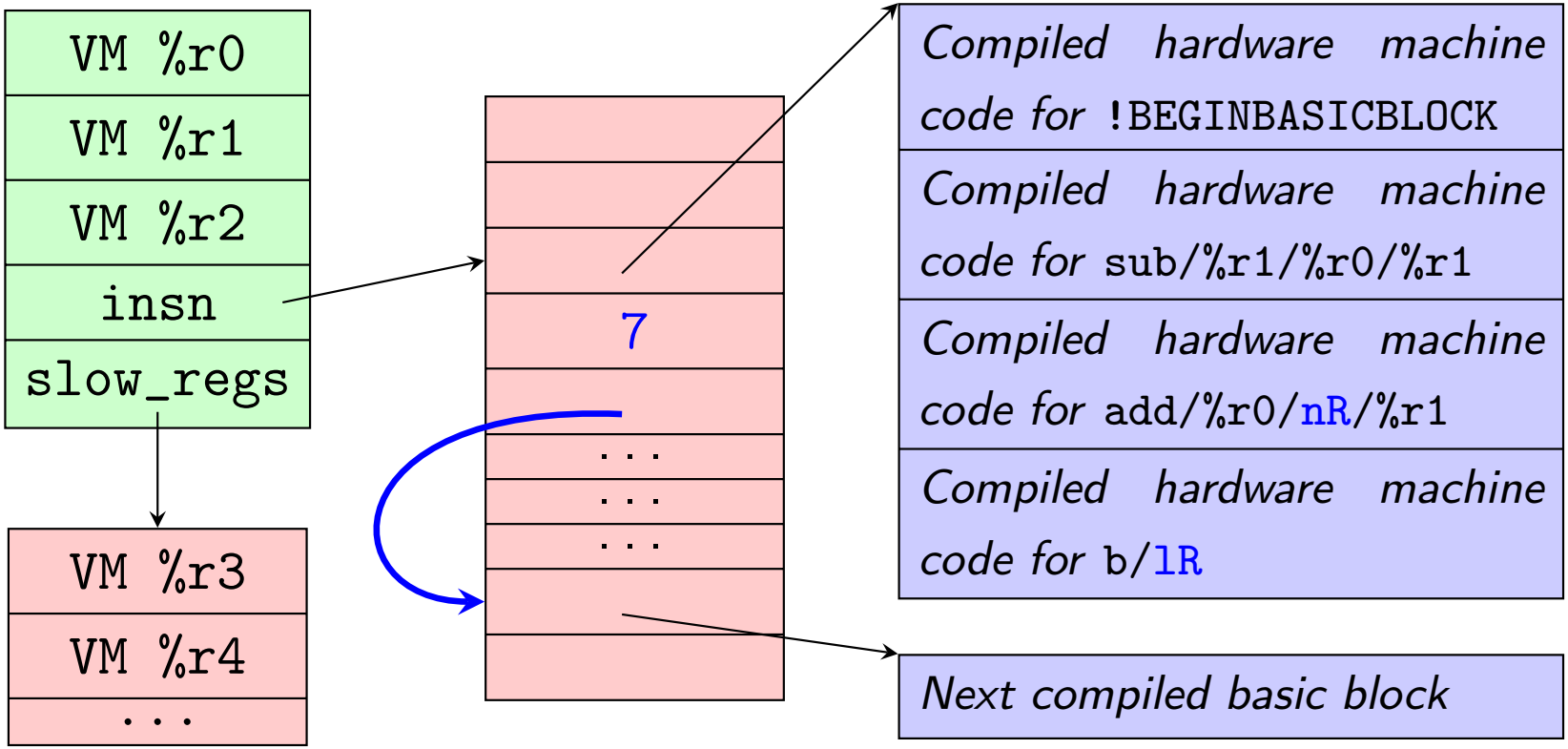
All VM instructions but unconditional branches end with slow fallthru code. We want to remove it.

The solution is copying compiled specialized VM instruction code sequences one after another, concatenating them into hardware machine-code basic blocks. Then each VM instruction in the block automatically "falls thru" into the next.

A code pointer is only needed at the beginning of each basic block.

I call this dispatching style minimal threading: it's an optimization of direct threading.

VM instruction replication example



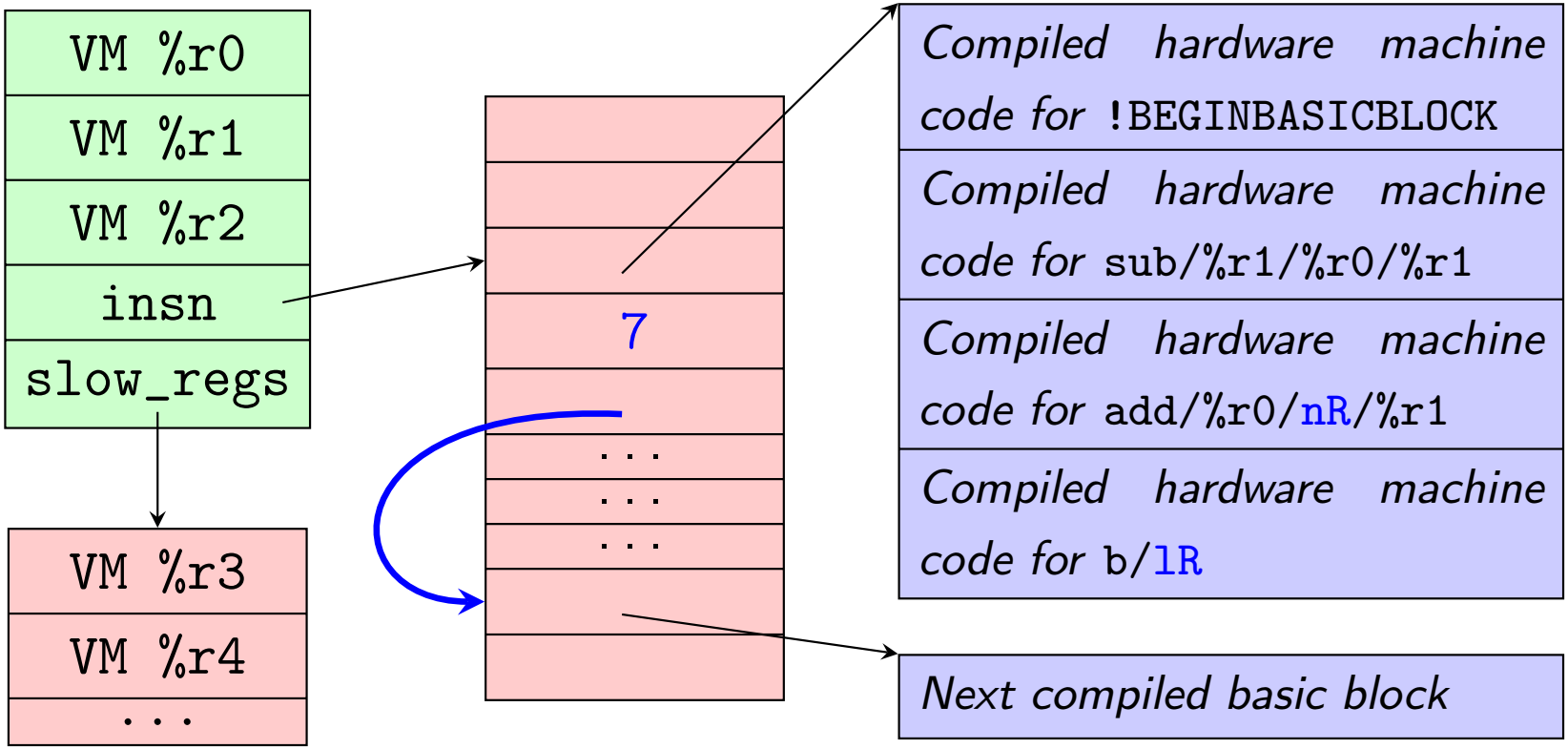
`!BEGINBASICBLOCK` only advances `insn` past the code pointer.

The `sub/%r1/%r0/%r1` VM instruction doesn't touch `L1d` or even `insn`.

The `add/%r0/nR/%r1` VM instruction has `7` as its residual literal argument on which it was not specialized.

Branch target arguments are not specialized for: the internal VM-program pointer is `b/lR`'s residual argument.

VM instruction replication example



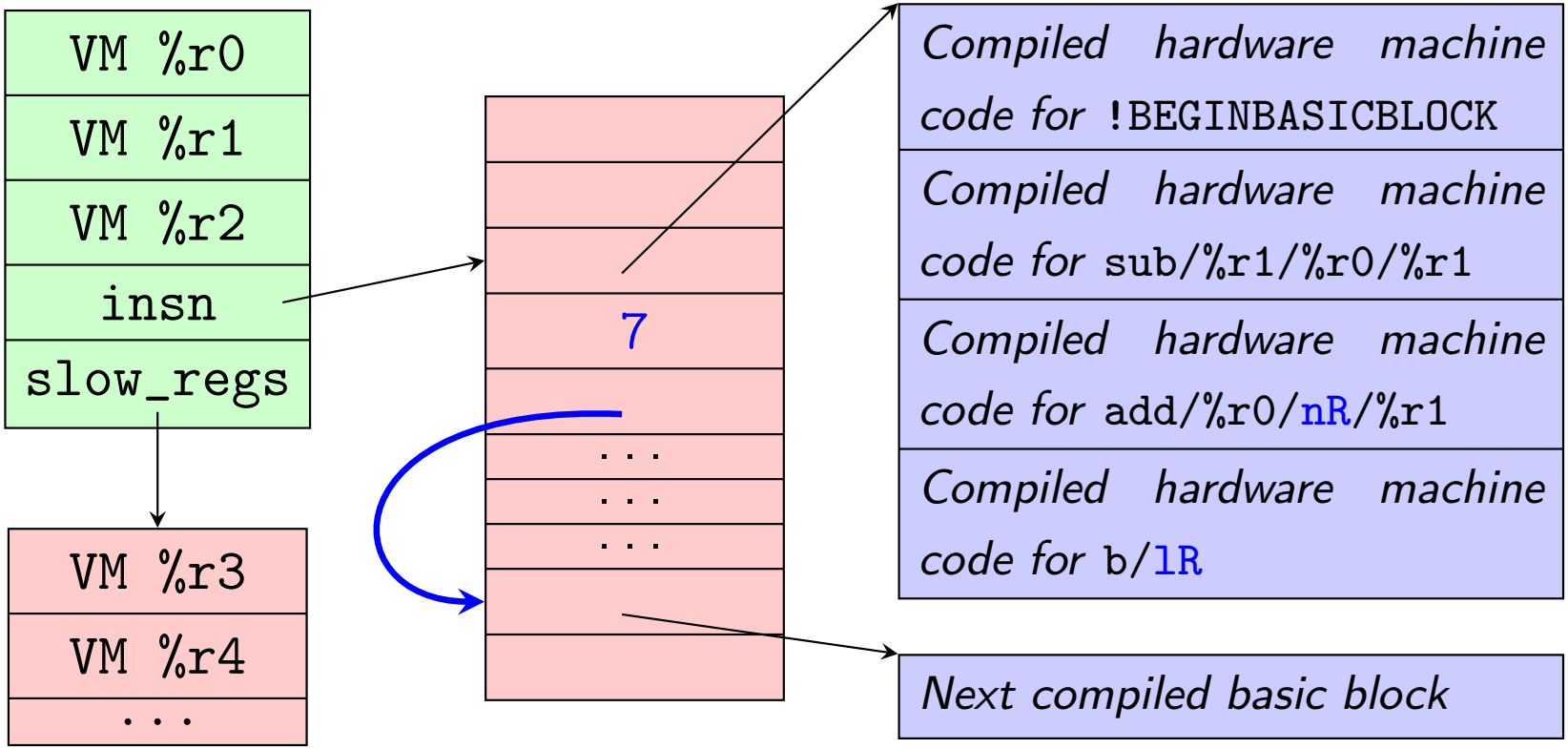
`!BEGINBASICBLOCK` only advances `insn` past the code pointer.

The `sub/%r1/%r0/%r1` VM instruction doesn't touch L1d or even `insn`.

The `add/%r0/nR/%r1` VM instruction has `7` as its residual literal argument on which it was not specialized.

Branch target arguments are not specialized for: the internal VM-program pointer is `b/lR`'s residual argument.

VM instruction replication example



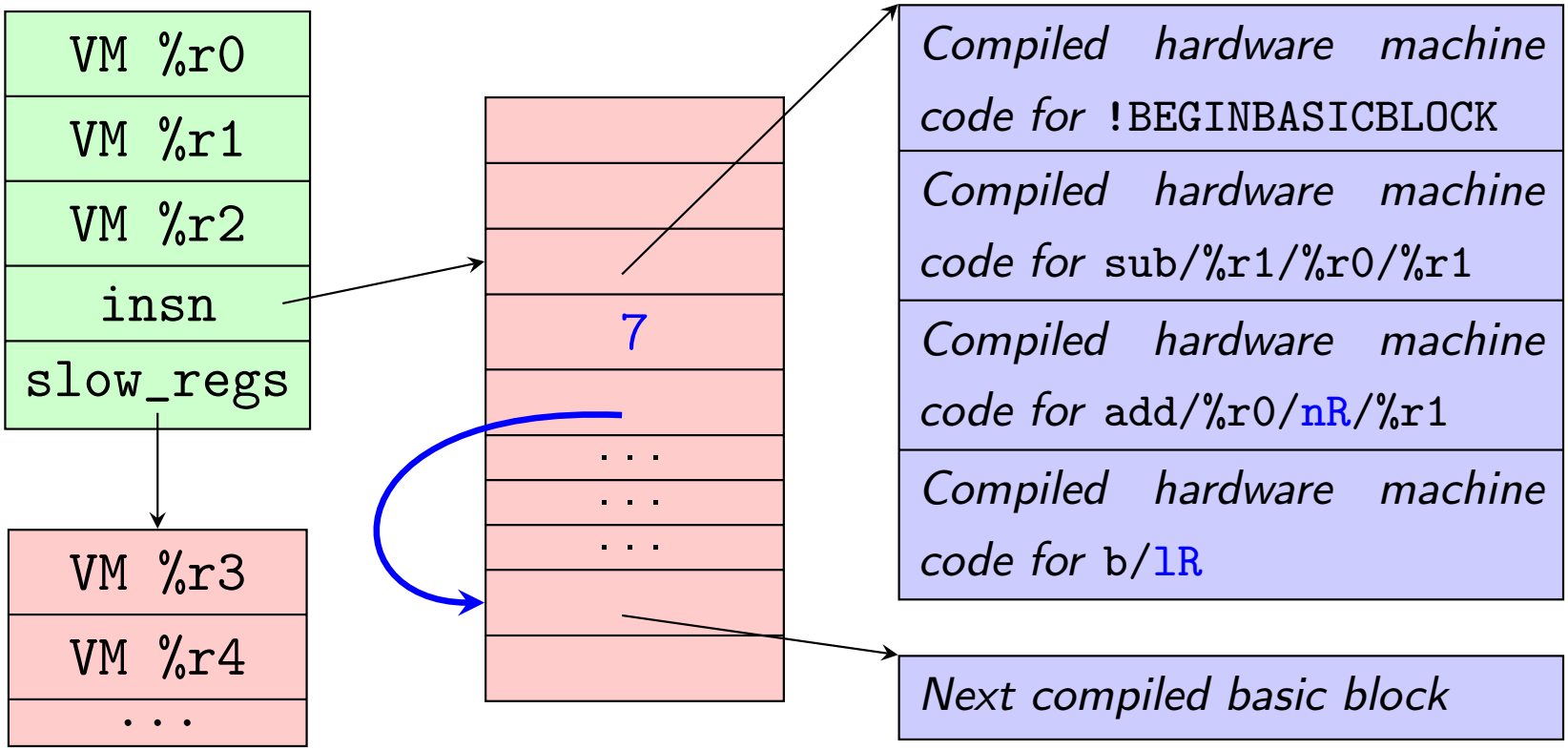
`!BEGINBASICBLOCK` only advances `insn` past the code pointer.

The `sub/%r1/%r0/%r1` VM instruction doesn't touch L1d or even `insn`.

The `add/%r0/nR/%r1` VM instruction has `7` as its residual literal argument on which it was not specialized.

Branch target arguments are not specialized for: the internal VM-program pointer is `b/lR`'s residual argument.

VM instruction replication example



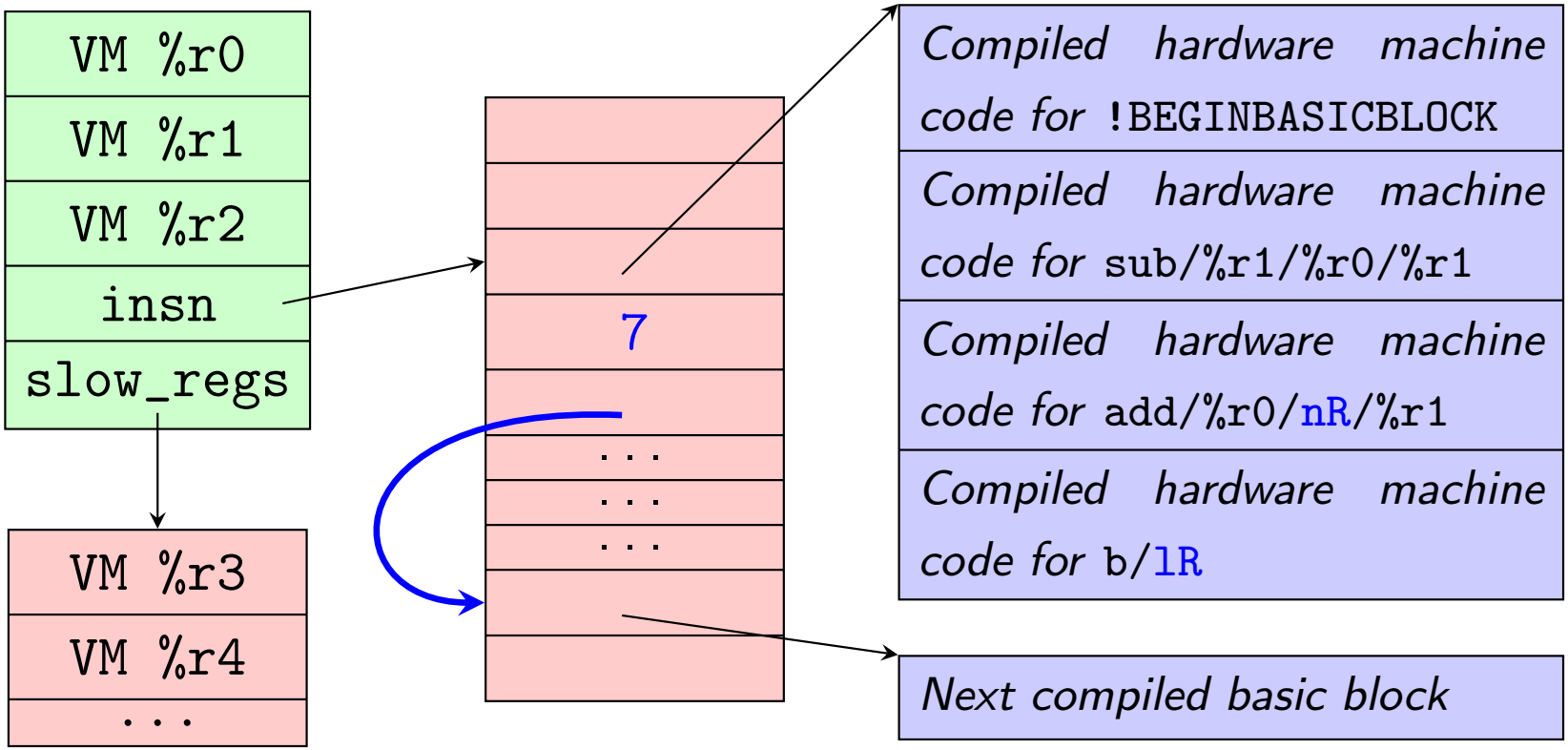
`!BEGINBASICBLOCK` only advances `insn` past the code pointer.

The `sub/%r1/%r0/%r1` VM instruction doesn't touch L1d or even `insn`.

The `add/%r0/nR/%r1` VM instruction has 7 as its residual literal argument on which it was not specialized.

Branch target arguments are not specialized for: the internal VM-program pointer is `b/lR`'s residual argument.

VM instruction replication example



`!BEGINBASICBLOCK` only advances `insn` past the code pointer.

The `sub/%r1/%r0/%r1` VM instruction doesn't touch L1d or even `insn`.

The `add/%r0/nR/%r1` VM instruction has 7 as its residual literal argument on which it was not specialized.

Branch target arguments are not specialized for: the internal VM-program pointer is `b/lR`'s residual argument.

VM instruction replication challenges

Replicating code by itself is not hard [but see Bruno's point on slide 60]:

- allocate executable memory with `mmap`
- copy machine code for VM specialized instructions into the executable space, delimited by label-as-value pointers.

We have to call GCC with the right options to prevent disasters:

- PC-relative memory accesses or calls.
- non-PIC code
- at least `-fno-reorder-blocks`, `-fpic` mandatory

More subtly, GCC needs to keep its register-allocation compatible across the code for every VM specialized instruction.

- a few tricks: jumps (unreachable in replicated code) at the end of specialized instruction code, jumping to a C jump with a destination unknown to GCC (`volatile`, no-code inline asm with constraints).

VM instruction replication challenges

Replicating code by itself is not hard [but see Bruno's point on slide 60]:

- allocate executable memory with `mmap`
- copy machine code for VM specialized instructions into the executable space, delimited by label-as-value pointers.

We have to call GCC with the right options to prevent disasters:

- PC-relative memory accesses or calls.
- non-PIC code
- at least `-fno-reorder-blocks`, `-fpic` mandatory

More subtly, GCC needs to keep its register-allocation compatible across the code for every VM specialized instruction.

- a few tricks: jumps (unreachable in replicated code) at the end of specialized instruction code, jumping to a C jump with a destination unknown to GCC (`volatile`, no-code inline asm with constraints).

VM instruction replication challenges

Replicating code by itself is not hard [but see Bruno's point on slide 60]:

- allocate executable memory with `mmap`
- copy machine code for VM specialized instructions into the executable space, delimited by label-as-value pointers.

We have to call GCC with the right options to prevent disasters:

- PC-relative memory accesses or calls.
- non-PIC code
- at least `-fno-reorder-blocks`, `-fpic` mandatory

More subtly, GCC needs to keep its register-allocation compatible across the code for every VM specialized instruction.

- a few tricks: jumps (unreachable in replicated code) at the end of specialized instruction code, jumping to a C jump with a destination unknown to GCC (`volatile`, no-code inline asm with constraints).

More VM instruction replication challenges

Global variable/function references are a problem (on most architecures), but given their names in C the generator can define macros to have them accessed thru a hidden stack-allocated structure — convenient for C code snippets.

VM specification

```
wrapped-globals
  printfixnum_format_string # String literals are dangerous!
end

wrapped-functions
  printf
  rand
  xmalloc
end
```

Since when replication is enabled we are already relying on another GCC extension we can afford `typeof` as well in the generated code, to free the user from the need of declaring types.

Minimal threading

Minimal threading is delicate but requires no assembly (unless `__builtin___clear_cache` fails to invalidate L1i, as I saw happen on powerpc).

Very portable: minimal threading is currently tested and working on aarch64, alpha, arm, i386, mips, powerpc, s390, sparc, x86_64 (either endianness, either bitness) — and it probably works on many more architectures. It currently fails on sh4, which relies heavily on PC-relative loads.

Minimal threading does require `mmap`, which isn't a problem on GNU systems. [After my talk Bruno Haible taught me a technique I didn't know for working around the restrictions of $W \oplus E$ systems, using *two mmap mappings*; still workable, but I admit that this will add some complexity]

A good dispatching model for most architectures. Where not supported (right now on sh4) the user can always revert to direct threading, lower-performance but as portable as GCC.

Minimal threading

Minimal threading is delicate but requires no assembly (unless `__builtin___clear_cache` fails to invalidate L1i, as I saw happen on powerpc).

Very portable: minimal threading is currently tested and working on aarch64, alpha, arm, i386, mips, powerpc, s390, sparc, x86_64 (either endianness, either bitness) — and it probably works on many more architectures. It currently fails on sh4, which relies heavily on PC-relative loads.

Minimal threading does require `mmap`, which isn't a problem on GNU systems. [After my talk Bruno Haible taught me a technique I didn't know for working around the restrictions of $W \oplus E$ systems, using *two mmap mappings*; still workable, but I admit that this will add some complexity]

A good dispatching model for most architectures. Where not supported (right now on sh4) the user can always revert to direct threading, lower-performance but as portable as GCC.

Minimal threading

Minimal threading is delicate but requires no assembly (unless `__builtin___clear_cache` fails to invalidate L1i, as I saw happen on powerpc).

Very portable: minimal threading is currently tested and working on aarch64, alpha, arm, i386, mips, powerpc, s390, sparc, x86_64 (either endianness, either bitness) — and it probably works on many more architectures. It currently fails on sh4, which relies heavily on PC-relative loads.

Minimal threading does require `mmap`, which isn't a problem on GNU systems. [After my talk Bruno Haible taught me a technique I didn't know for working around the restrictions of $W \oplus E$ systems, using *two mmap mappings*; still workable, but I admit that this will add some complexity]

A good dispatching model for most architectures. Where not supported (right now on sh4) the user can always revert to direct threading, lower-performance but as portable as GCC.

Minimal threading

Minimal threading is delicate but requires no assembly (unless `__builtin___clear_cache` fails to invalidate L1i, as I saw happen on powerpc).

Very portable: minimal threading is currently tested and working on aarch64, alpha, arm, i386, mips, powerpc, s390, sparc, x86_64 (either endianness, either bitness) — and it probably works on many more architectures. It currently fails on sh4, which relies heavily on PC-relative loads.

Minimal threading does require `mmap`, which isn't a problem on GNU systems. [After my talk Bruno Haible taught me a technique I didn't know for working around the restrictions of $W \oplus E$ systems, using *two mmap mappings*; still workable, but I admit that this will add some complexity]

A good dispatching model for most architectures. Where not supported (right now on sh4) the user can always revert to direct threading, lower-performance but as portable as GCC.

Minimal threading

Minimal threading is delicate but requires no assembly (unless `__builtin___clear_cache` fails to invalidate L1i, as I saw happen on powerpc).

Very portable: minimal threading is currently tested and working on aarch64, alpha, arm, i386, mips, powerpc, s390, sparc, x86_64 (either endianness, either bitness) — and it probably works on many more architectures. It currently fails on sh4, which relies heavily on PC-relative loads.

Minimal threading does require `mmap`, which isn't a problem on GNU systems. [After my talk Bruno Haible taught me a technique I didn't know for working around the restrictions of $W \oplus E$ systems, using *two mmap mappings*; still workable, but I admit that this will add some complexity]

A good dispatching model for most architectures. Where not supported (right now on sh4) the user can always revert to direct threading, lower-performance but as portable as GCC.

Next bottleneck: VM branches

With minimal threading we have mostly [we still need to increment `insn` for VM instructions with residual arguments] eliminated fallthru overhead.

The next bottleneck to eliminate is VM branching — fallthru overhead will also go away completely as a side effect.

- All VM branching overhead comes from the direct-threading convention of having VM program slots contain pointers to executable code.
- Moreover residual literals and slow register offsets are also loaded from the VM program in memory, which is usually suboptimal.

Why having the VM program as a data structure in memory at all?

Next bottleneck: VM branches

With minimal threading we have mostly [we still need to increment `insn` for VM instructions with residual arguments] eliminated fallthru overhead.

The next bottleneck to eliminate is VM branching — fallthru overhead will also go away completely as a side effect.

- All VM branching overhead comes from the direct-threading convention of having VM program slots contain pointers to executable code.
- Moreover residual literals and slow register offsets are also loaded from the VM program in memory, which is usually suboptimal.

Why having the VM program as a data structure in memory at all?

Next bottleneck: VM branches

With minimal threading we have mostly [we still need to increment `insn` for VM instructions with residual arguments] eliminated fallthru overhead.

The next bottleneck to eliminate is VM branching — fallthru overhead will also go away completely as a side effect.

- All VM branching overhead comes from the direct-threading convention of having VM program slots contain pointers to executable code.
- Moreover residual literals and slow register offsets are also loaded from the VM program in memory, which is usually suboptimal.

Why having the VM program as a data structure in memory at all?

Next bottleneck: VM branches

With minimal threading we have mostly [we still need to increment `insn` for VM instructions with residual arguments] eliminated fallthru overhead.

The next bottleneck to eliminate is VM branching — fallthru overhead will also go away completely as a side effect.

- All VM branching overhead comes from the direct-threading convention of having VM program slots contain pointers to executable code.
- Moreover residual literals and slow register offsets are also loaded from the VM program in memory, which is usually suboptimal.

Why having the VM program as a data structure in memory at all?

No-threading dispatch and and residual access

Introducing the last and most efficient dispatching mode, no threading.

The idea: do away with the VM problem as a data structure, and only keep the replicated executable code.

At this point we need some architecture-specific assembly code:

- Residual literals must be materialized into hardware registers or memory, since there is no program to load them from
  - Small hand-written assembly routines, to be patched with literals...
  - ...copied before the beginning of each VM specialized instruction code needing residuals.

No-threading dispatch and and residual access

Introducing the last and most efficient dispatching mode, no threading.

The idea: do away with the VM problem as a data structure, and only keep the replicated executable code.

At this point we need some architecture-specific assembly code:

- Residual literals must be materialized into hardware registers or memory, since there is no program to load them from
  - Small hand-written assembly routines, to be patched with literals...
  - ...copied before the beginning of each VM specialized instruction code needing residuals.

No-threading dispatch and and residual access

Introducing the last and most efficient dispatching mode, no threading.

The idea: do away with the VM problem as a data structure, and only keep the replicated executable code.

At this point we need some architecture-specific assembly code:

- Residual literals must be materialized into hardware registers or memory, since there is no program to load them from
  - Small hand-written assembly routines, to be patched with literals...
  - ...copied before the beginning of each VM specialized instruction code needing residuals.

No-threading dispatch and and residual access

Introducing the last and most efficient dispatching mode, no threading.

The idea: do away with the VM problem as a data structure, and only keep the replicated executable code.

At this point we need some architecture-specific assembly code:

- Residual literals must be materialized into hardware registers or memory, since there is no program to load them from
  - Small hand-written assembly routines, to be patched with literals...
  - ...copied before the beginning of each VM specialized instruction code needing residuals.

No-threading dispatch and and residual access

Introducing the last and most efficient dispatching mode, no threading.

The idea: do away with the VM problem as a data structure, and only keep the replicated executable code.

At this point we need some architecture-specific assembly code:

- Residual literals must be materialized into hardware registers or memory, since there is no program to load them from
  - Small hand-written assembly routines, to be patched with literals...
  - ...copied before the beginning of each VM specialized instruction code needing residuals.

No-threading branches

Without the VM program there is no longer need for `insn` either — not even in a hardware register.

- The VM instruction pointer is the same as the hardware instruction pointer (`%rip` on x86_64): native hardware branches!
  - Branching via a hardware register is easy in GNU C and requires no assembly: `goto *...`
  - ...but that would be suboptimal:

    `jmp L` is usually faster than `jmpq *%rax`.

No-threading branches

Without the VM program there is no longer need for `insn` either — not even in a hardware register.

- The VM instruction pointer is the same as the hardware instruction pointer (`%rip` on x86_64): native hardware branches!
  - Branching via a hardware register is easy in GNU C and requires no assembly: `goto *...`
  - ...but that would be suboptimal:

    `jmp L` is usually faster than `jmpq *%rax`.

No-threading branches

Without the VM program there is no longer need for `insn` either — not even in a hardware register.

- The VM instruction pointer is the same as the hardware instruction pointer (`%rip` on x86_64): native hardware branches!
  - Branching via a hardware register is easy in GNU C and requires no assembly: `goto *`...
  - ...but that would be suboptimal:

    `jmp L` is usually faster than `jmpq *%rax`.

No-threading branches

Without the VM program there is no longer need for `insn` either — not even in a hardware register.

- The VM instruction pointer is the same as the hardware instruction pointer (`%rip` on x86_64): native hardware branches!
  - Branching via a hardware register is easy in GNU C and requires no assembly: `goto *...`
  - ...but that would be suboptimal:

    `jmp L` is usually faster than `jmpq *%rax`.

No-threading dispatch: label arguments

Label literals, as wide constants, are painful to load on RISCs and also force the CPU to jump thru a register or memory.

- We want to replace jumps in C code snippets with the appropriate hardware machine instructions—also in the conditional case.
- difficult, as jumps may occur anywhere within compiled C code.
- solution: provide predefined macros `VMPREFIX_BRANCH_FAST`, `VMPREFIX_BRANCH_FAST_IF_LESS_THAN`, `VMPREFIX_BRANCH_AND_LINK_FAST`, ...

  expanding to *patch-ins*:

No-threading dispatch: label arguments

Label literals, as wide constants, are painful to load on RISCs and also force the CPU to jump thru a register or memory.

- We want to replace jumps in C code snippets with the appropriate hardware machine instructions—also in the conditional case.
- difficult, as jumps may occur anywhere within compiled C code.
- solution: provide predefined macros `VMPREFIX_BRANCH_FAST`, `VMPREFIX_BRANCH_FAST_IF_LESS_THAN`, `VMPREFIX_BRANCH_AND_LINK_FAST`, ...

  expanding to *patch-ins*:

No-threading dispatch: label arguments

Label literals, as wide constants, are painful to load on RISCs and also force the CPU to jump thru a register or memory.

- We want to replace jumps in C code snippets with the appropriate hardware machine instructions—also in the conditional case.
- difficult, as jumps may occur anywhere within compiled C code.
- solution: provide predefined macros `VMPREFIX_BRANCH_FAST`, `VMPREFIX_BRANCH_FAST_IF_LESS_THAN`, `VMPREFIX_BRANCH_AND_LINK_FAST`, ...

  expanding to *patch-ins*:

No-threading dispatch: label arguments

Label literals, as wide constants, are painful to load on RISCs and also force the CPU to jump thru a register or memory.

- We want to replace jumps in C code snippets with the appropriate hardware machine instructions—also in the conditional case.
- difficult, as jumps may occur anywhere within compiled C code.
- solution: provide predefined macros `VMPREFIX_BRANCH_FAST`, `VMPREFIX_BRANCH_FAST_IF_LESS_THAN`, `VMPREFIX_BRANCH_AND_LINK_FAST`, ...

  expanding to *patch-ins*:

What a patch-in is

Every *patch-in* use generates an sequence of `0x0`s in compiled code, of the right legnth for the missing hardware instruction(s) to be patched in — and add a pointer to the "hole" into a global table in a different assembly section, along with an id for the specialized instruction and the patch-in case (unconditional branch, branch-and-link, branch-if-less-than-zero...).

(Macro-expanded) GNU C, simplified

```
asm goto (".pushsection .data, 42\n"
          "    .quad hole_to_fill_%=\n"
          "    .quad " SPECIALIZED_INSTRUCTION_ID "\n"
          "    .quad " PATCH_IN_CASE "\n"
          ".popsection\n"
          "hole_to_fill_%=:\n"
          "    .skip " ROUTINE_LENGTH_IN_BYTES "\n"
          : : /* inputs... */
          : : unreachable_label_jumping_where_gcc_cant_know);
```

What a patch-in is

Every *patch-in* use generates an sequence of 0x0s in compiled code, of the right legnth for the missing hardware instruction(s) to be patched in — and add a pointer to the "hole" into a global table in a different assembly section, along with an id for the specialized instruction and the patch-in case (unconditional branch, branch-and-link, branch-if-less-than-zero...).

(Macro-expanded) GNU C, simplified

```
asm goto (".pushsection .data, 42\n"
          "    .quad hole_to_fill_%=\n"
          "    .quad " SPECIALIZED_INSTRUCTION_ID "\n"
          "    .quad " PATCH_IN_CASE "\n"
          ".popsection\n"
          "hole_to_fill_%=:\n"
          "    .skip " ROUNTINE_LENGTH_IN_BYTES "\n"
          : : /* inputs... */
          : : unreachable_label_jumping_where_gcc_cant_know);
```

Patch-ins in action

The assembly section containing the global table is scanned to compute the addresses to patch within replicated code.

Jumps generated this way, and some inline `asm` for conditional branches, can make VM branches optimal on a given architecture.

[Demo: disassembling and timing the down-counter under no-threading dispatch]

Patch-ins in action

The assembly section containing the global table is scanned to compute the addresses to patch within replicated code.

Jumps generated this way, and some inline `asm` for conditional branches, can make VM branches optimal on a given architecture.

[Demo: disassembling and timing the down-counter under no-threading dispatch]

What should I call this?

Am I still speaking of efficient interpreters, or have I already crossed into JIT territory? The answer may be blurry, particularly with respect to common public expectations.

I will avoid the question, and call the software a generator of efficient "virtual machines".

My VM generator is called Jitter, and a VM generated by Jitter will be "Jittery". You are free to follow your imagination in interpreting the name. Here are some possibilities:

- a software attempting to pass for a JIT without success
- a maker of JITs
- something shaky and unreliable

What should I call this?

Am I still speaking of efficient interpreters, or have I already crossed into JIT territory? The answer may be blurry, particularly with respect to common public expectations.

I will avoid the question, and call the software a generator of efficient "virtual machines".

My VM generator is called Jitter, and a VM generated by Jitter will be "Jittery". You are free to follow your imagination in interpreting the name. Here are some possibilities:

- a software attempting to pass for a JIT without success
- a maker of JITs
- something shaky and unreliable

What should I call this?

Am I still speaking of efficient interpreters, or have I already crossed into JIT territory? The answer may be blurry, particularly with respect to common public expectations.

I will avoid the question, and call the software a generator of efficient "virtual machines".

My VM generator is called Jitter, and a VM generated by Jitter will be "Jittery". You are free to follow your imagination in interpreting the name. Here are some possibilities:

- a software attempting to pass for a JIT without success
- a maker of JITs
- something shaky and unreliable

What should I call this?

Am I still speaking of efficient interpreters, or have I already crossed into JIT territory? The answer may be blurry, particularly with respect to common public expectations.

I will avoid the question, and call the software a generator of efficient "virtual machines".

My VM generator is called Jitter, and a VM generated by Jitter will be "Jittery". You are free to follow your imagination in interpreting the name. Here are some possibilities:

- a software attempting to pass for a JIT without success
- a maker of JITs
- something shaky and unreliable

What should I call this?

Am I still speaking of efficient interpreters, or have I already crossed into JIT territory? The answer may be blurry, particularly with respect to common public expectations.

I will avoid the question, and call the software a generator of efficient "virtual machines".

My VM generator is called Jitter, and a VM generated by Jitter will be "Jittery". You are free to follow your imagination in interpreting the name. Here are some possibilities:

- a software attempting to pass for a JIT without success
- a maker of JITs
- something shaky and unreliable

What should I call this?

Am I still speaking of efficient interpreters, or have I already crossed into JIT territory? The answer may be blurry, particularly with respect to common public expectations.

I will avoid the question, and call the software a generator of efficient "virtual machines".

My VM generator is called Jitter, and a VM generated by Jitter will be "Jittery". You are free to follow your imagination in interpreting the name. Here are some possibilities:

- a software attempting to pass for a JIT without success
- a maker of JITs
- something shaky and unreliable

The near future

I'm releasing Jitter's code right now, for the first time.

`http://ageinghacker.net/ghm-2017`

There are rough edges but the code is not terrible. If you like languages you'll have fun.

- I want to propose Jitter as a GNU project.
- Implementation-wise, rewrite rules are the most urgent thing.
  [I also have to actually use the Array; that's easy and will be ready soon, possibly before the GHM is over. Hierarchical wrapped globals will have to wait a little.]
- I have to finish the manual. Of the already existing part I strongly recommend the section about when *not* to use VMs in the introduction.

The near future

I'm releasing Jitter's code right now, for the first time.

`http://ageinghacker.net/ghm-2017`

There are rough edges but the code is not terrible. If you like languages you'll have fun.

- I want to propose Jitter as a GNU project.
- Implementation-wise, rewrite rules are the most urgent thing.
  [I also have to actually use the Array; that's easy and will be ready soon, possibly before the GHM is over. Hierarchical wrapped globals will have to wait a little.]
- I have to finish the manual. Of the already existing part I strongly recommend the section about when *not* to use VMs in the introduction.

The near future

I'm releasing Jitter's code right now, for the first time.

`http://ageinghacker.net/ghm-2017`

There are rough edges but the code is not terrible. If you like languages you'll have fun.

- I want to propose Jitter as a GNU project.
- Implementation-wise, rewrite rules are the most urgent thing.
  [I also have to actually use the Array; that's easy and will be ready soon, possibly before the GHM is over. Hierarchical wrapped globals will have to wait a little.]
- I have to finish the manual. Of the already existing part I strongly recommend the section about when *not* to use VMs in the introduction.

The near future

I'm releasing Jitter's code right now, for the first time.

`http://ageinghacker.net/ghm-2017`

There are rough edges but the code is not terrible. If you like languages you'll have fun.

- I want to propose Jitter as a GNU project.
- Implementation-wise, rewrite rules are the most urgent thing.
  [I also have to actually use the Array; that's easy and will be ready soon, possibly before the GHM is over. Hierarchical wrapped globals will have to wait a little.]
- I have to finish the manual. Of the already existing part I strongly recommend the section about when *not* to use VMs in the introduction.

Thank you

Also thanks to the people from whose work I learned the bases on which I built Jitter, particularly Anton Ertl. See the bibliography on slide 70, and the `NOTES` file in the tarball.

My virtual machine is faster than yours.

Any questions?

Are you thinking of some application for Jitter? Tell me.

Bibliography I


Ertl, M. A. (2008). The Vmgen manual. The manual is in Texinfo, distributed along with GForth. Do a `M-x info vmgen` if you use the Emacs Info reader.


Ertl, M. A. and Gregg, D. (2004). Retargeting JIT compilers by using C-compiler generated executable code. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 41–50, Washington, DC, USA. IEEE Computer Society.


Ertl, M. A., Gregg, D., Krall, A., and Paysan, B. (2002). Vmgen – a generator of efficient virtual machine interpreters. *SoftwarePractice and Experience*, 32:2002.

Bibliography II

Saiu, L. (2017). The Jitter `NOTES` file. The `NOTES` file in the current Jitter distribution contains my (crudely) annotated bibliography, originally intended just for myself, with many more references. Not really a literature review, but at least a list of useful pointers to scientific publications.

Shi, Y., Gregg, D., Beatty, A., and Ertl, M. A. (2005). Virtual machine showdown: Stack versus registers. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 153–163, New York, NY, USA. ACM. There exists a 2008 paper with the same title, similar abstract and almost the same authors, clearly reporting new developments; I haven't found a copy. Yunhe Shi's PhD thesis from 2007 is also closely related, and arrives at the same conclusions.